Applying Machine Learning to Root Cause Analysis in Agile CI/CD Software Testing Environments

Julen Kahles Bastida

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Helsinki, Finland, 20.12.2018

Supervisor

Prof. Alexander Jung

Advisors

Lic.Sc. (Tech.) Timo Huuhtanen

D.Sc. (Tech.) Jukka Ylitalo

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Julen Kahles Bastida

Title Applying Machine Learning to Root Cause Analysis in Agile CI/CD Software Testing Environments

Degree programme Computer, Communication and Information Sciences

Major Acoustics and Audio Technology **Code of major** ELEC3030

Supervisor Prof. Alexander Jung

Advisors Lic.Sc. (Tech.) Timo Huuhtanen, D.Sc. (Tech.) Jukka Ylitalo

Date 20.12.2018 **Number of pages** 79 **Language** English

Abstract

This thesis evaluates machine learning classification and clustering algorithms with the aim of automating the root cause analysis of failed tests in agile software testing environments. The inefficiency of manually categorizing the root causes in terms of time and human resources motivates this work. The development and testing environments of an agile team at Ericsson Finland are used as this work's framework. The author of the thesis extracts relevant features from the raw log data after interviewing the team's testing engineers (human experts). The author puts his initial efforts into clustering the unlabeled data, and despite obtaining qualitative correlations between several clusters and failure root causes, the vagueness in the rest of the clusters leads to the consideration of labeling. The author then carries out a new round of interviews with the testing engineers, which leads to the conceptualization of ground-truth categories for the test failures. With these, the human experts label the dataset accordingly. A collection of artificial neural networks that either classify the data or pre-process it for clustering is then optimized by the author. The best solution comes in the form of a classification multilayer perceptron that correctly assigns the failure category to new examples, on average, 88.9% of the time. The primary outcome of this thesis comes in the form of a methodology for the extraction of expert knowledge and its adaptation to machine learning techniques for test failure root cause analysis using test log data. The proposed methodology constitutes a prototype or baseline approach towards achieving this objective in a corporate environment.

Keywords root cause analysis, software testing, log data analysis, machine learning, artificial neural networks, classification, clustering, automation

Wayfarer, the only way
is your footsteps, there is no other.
Wayfarer, there is no way,
you make the way by walking.
As you go, you make the way
and stopping to look behind,
you see the path that your feet
will never travel again.
Wayfarer, there is no way –
only foam trails to the sea.

Antonio Machado (1875–1939)
Campos de Castilla, "Proverbios y cantares" (XXIX)
Translation by Alan S. Trueblood (1917–2012)

Preface

This thesis was carried out at Ericsson Finland under the supervision of Prof. Alexander Jung (Aalto University), Lic.Sc. (Tech.) Timo Huuhtanen (Aalto University), D.Sc. (Tech.) Jukka Ylitalo (Ericsson Finland), M.Sc. Jarno Kyykkä (Ericsson Finland), and Juha Törrönen (Ericsson Finland).

I could not be more beholden to all the people who have helped me during this time, to whom I would like to express my gratitude:

To my mother Marta, for her faith in me, and for supporting me with my decision to pursue Master's studies in Finland.

To Alex and Timo, for their guidance in the development of this project, and for the insightful conversations that took place both in Otaniemi and Jorvas.

To my colleagues from Ericsson Finland, and especially, Jukka, Jarno, Juha, Juha, Adam, Ilmo, Mikko, and Zakaria, for the good advice and help that I received.

To Luis, for all the engaging discussions we had on LaTeX.

To Jorge, for being the one who guided me towards becoming an engineer.

Without your support, this journey would not have been possible.

Thank you.

Helsinki, Finland, 20.12.2018

Julen Kahles Bastida

Contents

List of Figures

List of Tables

List of Listings

Notation

Elements

x	A scalar
$\boldsymbol{x}$	A vector
$\boldsymbol{X}$	A matrix
$\boldsymbol{X}^T$	Transpose of matrix $\boldsymbol{X}$
x_i	Element i of vector $\boldsymbol{x}$
$X_{i,j}$	Element (i, j) of matrix $\boldsymbol{X}$
$\boldsymbol{X}_{i,:}$	Row i of matrix $\boldsymbol{X}$
$\boldsymbol{X}_{:,j}$	Column j of matrix $\boldsymbol{X}$
$x^{(t)}$	Scalar x at algorithm iteration t
$\boldsymbol{x}^{(t)}$	Vector $\boldsymbol{x}$ at algorithm iteration t
X	A random variable
$\mathbb{X}$	A set
$\{x_1, x_2, \ldots, x_n\}$	A set containing all elements from x_1 to x_n
$[a, b]$	The real interval including a and b

Symbols and operators

E	Expected value of a random variable
H	Information or Shannon entropy
I	Self-Information
σ	Standard deviation of a random variable
σ^2	Variance of a random variable
Σ	Covariance matrix of a random vector
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$
∇	Gradient of a function
$\lvert\boldsymbol{x}\rvert$	Number of elements in vector $\boldsymbol{x}$
$\lVert\boldsymbol{x}\rVert_p$	L^p norm of vector $\boldsymbol{x}$
$\cap$	Intersection of two sets

Abbreviations

Adam	Adaptive Moment Estimation
AE	Autoencoder
AI	Artificial Intelligence
AMI	Adjusted Mutual Information
ANN	Artificial Neural Network
ARI	Adjusted Rand Index
CI/CD	Continuous Integration/Continuous Deployment
DL	Deep Learning
GD	Gradient Descent
GMM	Gaussian Mixture Model
HTTP	Hypertext Transfer Protocol
MI	Mutual Information
ML	Machine Learning
MLP	Multilayer Perceptron
MSE	Mean Squared Error
MVP	Minimum Viable Product
NMI	Normalized Mutual Information
OS	Operating System
PCA	Principal Component Analysis
RCA	Root Cause Analysis
ReLU	Rectified Linear Unit
RI	Rand Index
RL	Representation Learning
RV	Random Variable
SGD	Stochastic Gradient Descent
tanh	Hyperbolic tangent
XP	Extreme Programming

1 Introduction

1.1 Background and motivation

The digitization of our human society and the rapid and steady progress of hardware devices in the last decades has caused software to become ubiquitous in virtually every aspect of our lives: software is now an essential and indispensable engine for our society to function [1, 2, 3, 4]. The importance of the software industry is steadily growing, and along with it, the complexity of the produced software systems [5, 6].

In addition to its development, software needs to be tested and debugged so as to ensure it provides its intended functionality in a flawless fashion [7, 6]. Automated tests generate raw diagnostics in the form of log files: these record everything that is tested and the results the tests produce, tracking, among others, variable values, function calls, inputs, and outputs [8, 9]. When tests fail, testing engineers need to manually analyze the log data to determine the origin of the failures and gain insights for code correction [10, 11]. This activity is tackled via root cause analysis (RCA), a process defined as a structured investigation of a problem aimed at pinpointing its cause, obtaining feedback for improvement and future error prevention [12, 10, 11].

Alongside the growing intricacy of software projects, the task of testing and debugging software is becoming more complicated and expensive, given the difficulty for testing engineers to keep track of all pieces of code that are developed in a project [13, 6]. This is further stressed in modern agile environments, where software is aimed at being made ready for release in fast-paced sprints (i.e., completion periods). In such environments, the application of continuous integration and deployment (CI/CD) has become a common practice [14]. Firstly, as soon as new code is developed, it is automatically tested, which constitutes the first action of CI. Should any tests fail, the developer in charge will need to update the code, correcting any possible bugs. Once this iterative cycle results in all the tests passing, the code is pushed to the main branch (staging or production system), which constitutes the second action of CI: the test results control whether the developed software is ready for being included in the final production system. Ensuring that the code in the main branch is ready for release at any time corresponds to the practice of CD [15, 14]. As a result of this process, vast amounts of log files are generated in short time intervals, whose unstructured formatting is unintuitive and abstruse to human readers [16].

In order to meet the tight deployment-cycle time constraints, when analyzing log files manually, testing engineers generally rely on their intuition and accumulated knowledge, checking only what are deemed to be suspicious log files. Usually, these files are queried with search commands (such as `grep`) for specific predefined keywords (e.g., `error`, `fail`, `crash`, etc.) [16].

It is a well-known fact that software outperforms humans in analyzing vast volumes of log data in terms of speed and cost: besides their fast processing capabilities, machines do not get tired of analyzing log files, and their performance is also unaffected by tedium or distraction.

Hence, given the expensive and potentially automatable action performed by testing engineers, manual RCA proves to be time-consuming, costly, and overall, an inefficient process [17].

As a first option for automating RCA, the log files can be analyzed by a rule-based system consisting of a software program whose behavior is fully enclosed by its coded specifications [16]. This presents itself as a solution for overcoming the inefficient nature of manual RCA; nonetheless, it faces a major disadvantage that makes its use suboptimal [16, 18]: the sheer amount of conditions and statements that need to be hard-coded causes the proposal to degenerate into an impractical solution, where certain failure conditions might even be unknown and not originally accounted for until they take place.

Machine learning (ML) excels in such a scenario: the good performance it obtains over vast amounts of data without the need for strictly defined rules makes it an appealing option [19]. A working ML solution learns from the data and can potentially find hidden patterns not accounted for by either manual labor or a rule-based software solution: it presents itself as the most effective and economical choice.

1.2 Purpose

Given the background and problem statement presented in Section 1.1, this work pursues an ML system for automating test failure RCA via log data analysis in agile CI/CD software testing environments.

The obtained results serve as a proof of concept or baseline approach towards automating test failure RCA in the corporate testing environment of an agile team at Ericsson Finland. The primary result of this work comprises a methodology for the extraction of expert knowledge and its adaptation to ML techniques for RCA: the stages of feature extraction, conceptualization of ground-truth categories, and algorithm assessment are tackled. The outcome presented in this work constitutes a prototype solution based on the obtained findings.

This Master's thesis has been funded by Ericsson Finland and has been carried out as a research project within its R&D division.

1.3 Research method

The following research questions are addressed in this work:

1. How can meaningful features be defined and extracted from software testing log data?
2. How can the existing developers' knowledge be mapped into distinctive ground-truth root-cause classes?
3. How can the root cause of a failed test be identified using ML?

In order to answer them, the author follows the research methodology of action research [20, 21].

In its first phase of inquiry, the author poses the initial research questions, and a literature review follows in order for him to gain an understanding of the state of the art. Once this is accomplished, the design process takes place, where the author sets short-term objectives with illustrative deadlines.

In the second phase of action, the author collects data in the form of features, and experimentation happens, where the author drafts and deploys different ML solutions. The author assesses these methods by selecting specific metrics, studied in the literature review.

Afterward, in the phase of analysis, the author collects the results. The author iterates between these three phases until the desired outcome is obtained and all research questions can be answered satisfactorily. Once this is achieved, the author reaches the final conclusions and reports them.

The whole iterative process is illustrated in Figure 1.

Regarding the development of ML algorithms, the author follows the principles of the agile programming methodology, which are aimed at iteratively developing minimum viable products (MVPs) in short-term paces, gathering rapid feedback and updates on how to steer the direction of the development [22]. Due to the individual nature of this work, the author carried out weekly meetings with the thesis supervisors (who acted as product owners), where the evolution and status of the project were discussed, and feedback was obtained.

The choice of following the principles of the agile methodology gets further endorsed by the fact that it is the methodology used in the development and testing environments of Ericsson Finland.

1.4 Structure of the thesis

The remainder of this thesis is organized as follows: Section 2 reviews the previous work related to the objective of applying ML to the automation of software testing failure RCA; Section 3 describes the testing environment this work is based on; Section 4 tackles the used ML technologies in terms of their analytical definition and functionality; Section 5 presents the development work, the implemented architectures, and the design criteria that lead to the final proposals; Section 6 gathers the obtained results; Finally, Section 7 concludes this thesis.

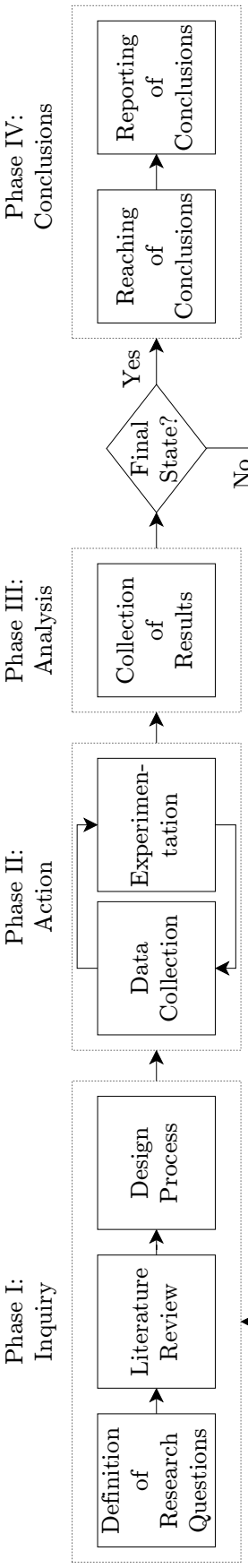
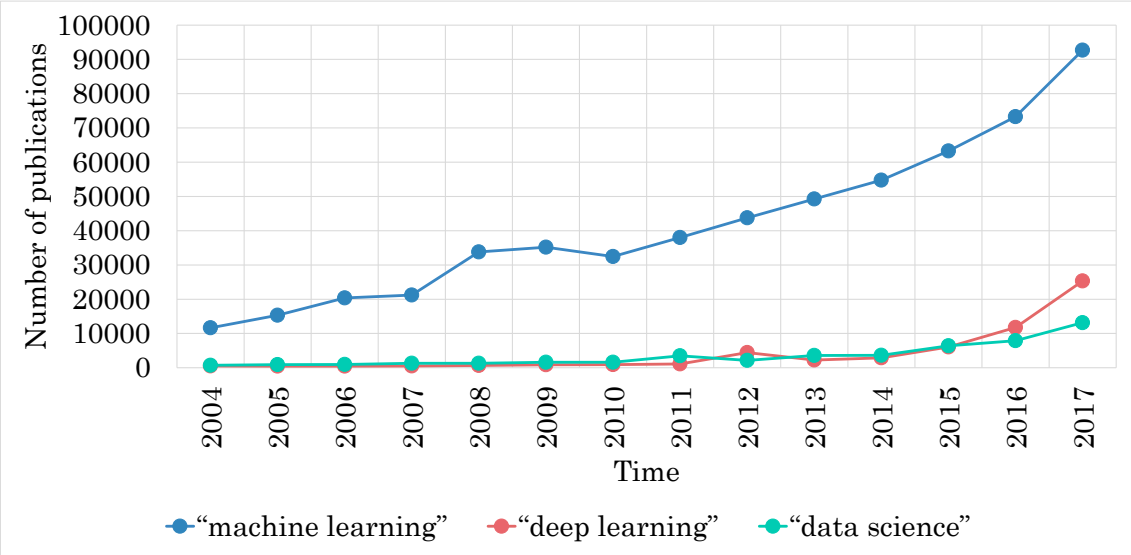


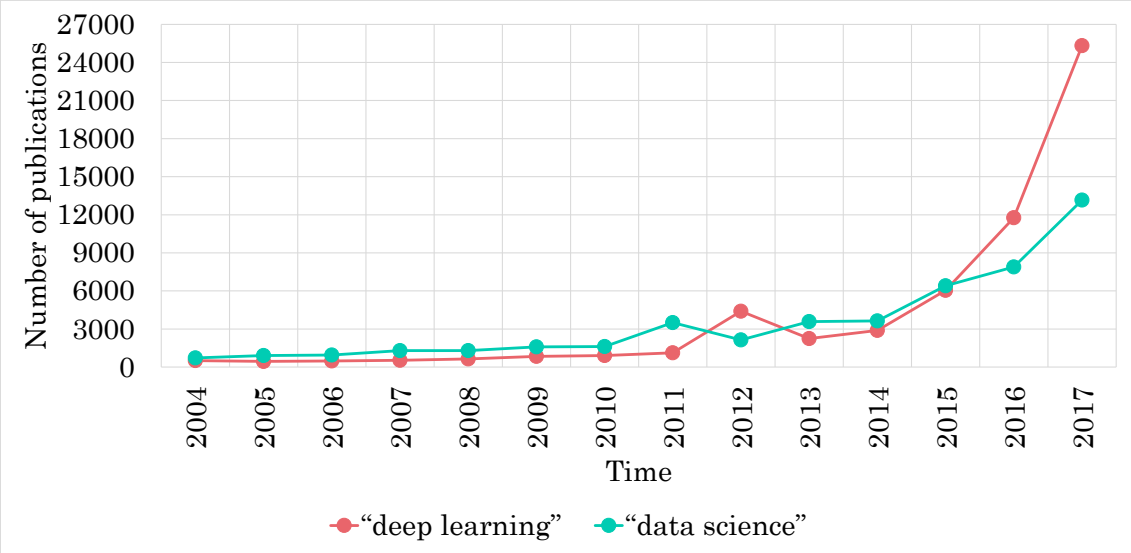
Figure 1: Block diagram depicting the phases and iterative nature of the action research methodology [20, 21].

2 State of the art

In recent years, the disciplines of ML, deep learning (DL), and data science have grown significantly in importance in both corporate environments and academia [23, 24], where the number of published research works including any of these terms has been steadily increasing on a yearly basis, as can be seen in Figure 2.



(a) Time evolution of the number of publications containing the terms "machine learning", "deep learning", and "data science".



(b) Zoom of Figure 2a to the terms "deep learning" and "data science".

Figure 2: Time evolution of the worldwide count of publications containing the terms "machine learning", "deep learning", and "data science" [25].

Many factors have contributed to this phenomenon [26, 23]: firstly, the advances in computational power have caused powerful hardware to be affordable, which is needed to process the vast amounts of complex information that ML systems require to excel; additionally, influenced by the generalized use of the Internet, the availability of datasets is rising, some of which are even available free of charge; on top of this, extensive research on the academic field of ML has caused its theoretical understanding to increase, leading to new algorithms and refinements of mature proposals that enhance its overall performance; lastly, a vast amount of ML software development tools and libraries have been emerging, many of which are offered as free software.

Along these lines, and given the opportunity of exploiting the capabilities of ML in a wide range of applications, an extensive effort has been put into applying ML and statistical techniques to software testing [27, 28, 29, 30, 31, 32]: this situation can be further appreciated in Figure 3, which shows how research in this field is continually growing.

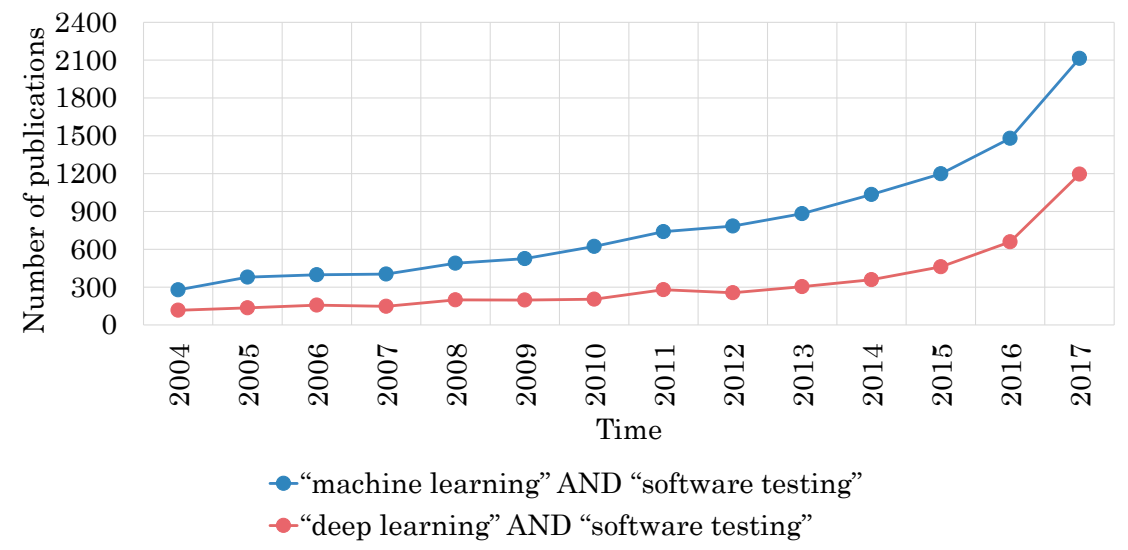


Figure 3: Time evolution of the worldwide count of publications containing the terms "machine learning" and "deep learning" combined with "software testing" in their abstracts [33].

When dealing specifically with log data analysis, different techniques have been researched.

An early attempt was carried out by Andrews in 1998 [8]: he presents finite-state machine design criteria for analyzing log files automatically. An automaton designed in such a way analyzes the events that happen as a program runs, tracked in the produced log files, and does not "learn" in an ML-sense, causing its action to fall under the category of artificial intelligence (AI; the distinction between the two disciplines is clarified in Section 4): the activity of the log analyzer consists on stating whether an analyzed log file conforms to a given specification or not.

Regarding pattern recognition, Weiss and Hirsh [34] present *timeweaver*, an ML system that makes use of genetic algorithms for identifying rare events in sequential data: past events are used to predict a current event, which is deemed to be "rare" if the prediction does not match it. Vaarandi [35] investigates a clustering algorithm based on log word frequency: it is designed to detect word clusters in log messages so that each cluster corresponds to a particular line pattern that occurs frequently enough. His envisioned algorithm is released via the *Simple Logfile Clustering Tool*.

Concerning failure prediction, Fulp et al. [36] present a spectrum-kernel-based support vector machine to predict software failure events based on system log files: the frequency representation of sequences of system log messages is fed to a support vector machine using a sliding window. Taking a different route, in his doctoral dissertation, Salfner [37] introduces a continuous-time extension of hidden Markov models applied to event-driven time sequences of errors: it is built on the recognition of symptomatic patterns of error sequences. Fronza et al. [38] continue the work on support vector machines, combining them with random indexing: the former get fed with sequences of operations extracted from log files, obtained by the action of the latter.

With respect to anomaly detection, Lee [39] presents his research on the field of inductive learning, exploring the application of data mining techniques for building intrusion detection models. Xu et al. [40] investigate the use of principal component analysis (PCA) combined with term-weighting on parsed console logs.

Commercially, several applications have been developed in recent years serving the need of log data analysis, like the highly successful Splunk and Elasticsearch-Logstash-Kibana Stack, among others [41, 42, 43]. The interest in them has grown in recent years, as can be appreciated in Figure 4. Given this situation, several researchers have also focused their attention on commercially available tools, like Stearly et al. [44], who study the effectiveness of Splunk in simplifying data mining tasks, showing how such a tool greatly facilitates extracting valuable information from machine data.

With regard to the topic of applying ML to the task of identifying the root causes of failed tests using log data analysis, previous research has framed this scenario as either an anomaly detection problem or an abnormal log detection issue.

Along these lines, Stearley [46] pursues a system analyzer by tackling system log messages. In his work, a comparison between the performance of a bioinformatic-inspired algorithm, known as *TEIRESIAS* [47], and the previously mentioned *Simple Logfile Clustering Tool* by Vaarandi [35] takes place: the former performs a useful log analysis by detecting anomalies and investigating cause-effect hypothesis at the cost of a non-scalable system memory demand, whereas the latter provides less effective results based on detecting word clusters in log messages without any memory restriction, being able to tackle longer log files (that exceed 10000 lines). In a recent publication, Du et al. [48] develop *DeepLog*, a deep neural network model for anomaly detection and diagnosis of system logs, modeled as natural language

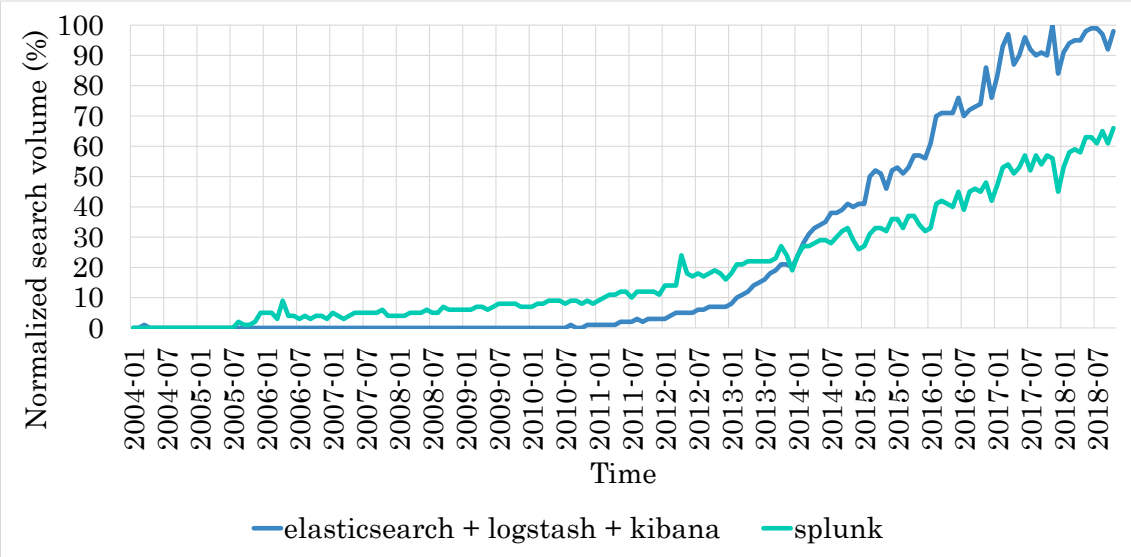


Figure 4: Worldwide normalized Google Trends search volume of the Splunk and ELK Stack (Elasticsearch, Logstash, Kibana) log analyzers [45, 41].

sequences. *DeepLog* is based on a long short-term memory recurrent neural network: it is trained to learn patterns during normal execution and detects anomalies when the obtained patterns differ from the learned ones. Lastly, Debnath et al. [49] present *LogLens*, a real-time log anomaly detector that works with minimal or even no target system knowledge and user specification, based on unsupervised learning: it learns the normal behavior of log events, builds a finite-state machine that captures it, and afterward, makes use of it for detecting anomalies.

Recent work in the form of a Master's thesis carried out at Ericsson gets closer to the topic of this thesis: Felldin [50] tackles log data collected from Ericsson Base Stations with the aim of obtaining the cause of a fatal error, implementing a Bayesian classifier for this purpose. Despite its closeness with this work, his study focuses on the specific Ericsson Base Stations' setting, tackling dump file data rendered as markup files and using specific ground-truth error classes from malfunctioning base stations: neither does the environment correspond to a more general agile CI/CD testing setting, nor can the methodology and results be generalized to one.

Therefore, the review of the existing literature this section has carried out indicates that no previous work has been published analyzing the performance of clustering and classification algorithms for determining the root causes of failed tests in agile CI/CD testing environments.

3 Software testing in agile software engineering

3.1 The paradigms of agile, extreme programming, and CI/CD

Before the 1990s, software engineering practices generally focused on thoughtful and extensive planning, heavy and rigorous control of the development, and formalized quality assurance [4]. Whereas these practices worked well in large, long-lasting, and critical software projects, such as the ones from the aerospace industry or even governmental systems, they generated discontent in small- and medium-sized corporations involving projects that had shorter time spans and needed more prototyping, as more time and resources were spent on the planning stages than on the actual development. Furthermore, specification reviews and changes were unable to be tackled at a quick pace, due to the theoretical need for planning before coding.

This situation led to the conception of the agile methodology, which gives more importance to the actual product development than to its design and documentation [22]. Overall, the agile methodology in any of its variations could be summarized as "people over process", stressing the objective of reducing bureaucracy. It aims at avoiding work steered in a direction that might not be, after all, the desired one, a strict and rigid hierarchy which slows down the pace of the development, and the writing of extensive documentation that might never be used.

Since its envisioning, several variations of the agile methodology have been proposed; among them, extreme programming (XP) is characterized by a fast-paced, iterative cycle of the software development [4].

In a general fashion, its target is to adapt the software development process to potential changes in stakeholder requirements: these are expressed as scenarios (user stories), from which a granular division into tasks is obtained. For each task, a series of tests are conceived and coded before their actual development takes place; after this has been achieved, the required code is produced and tested. Once all tests result in success, the produced code is officially integrated into the system. The completion of all tasks leads to the release of a new version of the software program, followed by an overall evaluation of the system. The feedback that is obtained after this stage triggers a new iteration in the overall cycle, which is illustrated in Figure 5.

The iterations are carried out in short periods of time, where frequent releases ensure an incremental development that pursues individual MVPs (i.e., working programs that feature some sought functionality): larger objectives are divided into smaller ones that can be implemented and tested within one iteration, and each task is assigned to a small team of approximately 3 to 9 programmers who tackle the development specified from a user's perspective. The purpose of this is to provide the project's stakeholders with a continuous prototype to gather feedback and be able to decide how to act next, instead of working for a long time on a product that might eventually not be satisfactory.

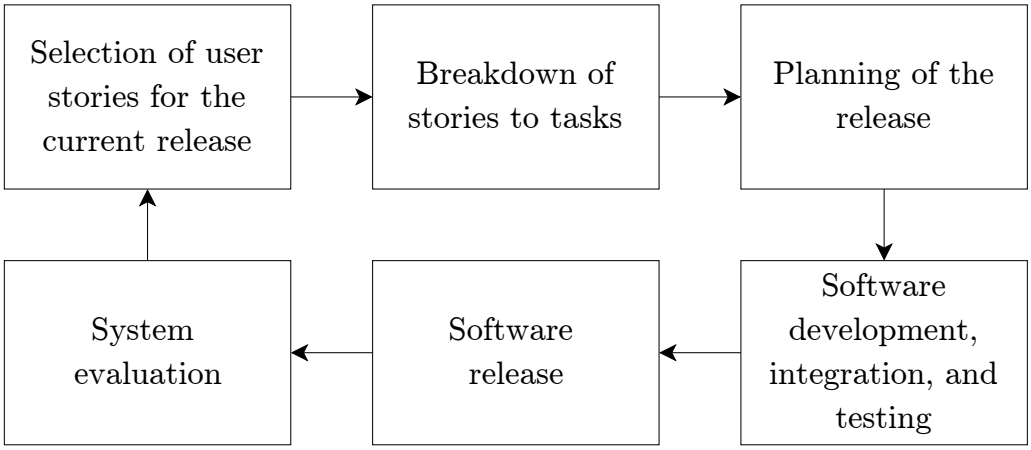


Figure 5: Iterative workflow of the XP methodology. Figure adapted from [4].

Throughout the process, communication and cooperation between the developers are pursued via daily meetings and informal conversations. The stakeholders are openly invited to engage in the development process, steering its direction: change is not only accepted but also embraced.

One of the core principles of XP is that of CI/CD. As soon as new code is generated, it is tested (CI). If any test fails, the code needs to be fine-tuned by the developer in charge; once all tests are passed, the code is integrated to the system, being generated ready for release at any moment, however mature the implementation might be (CD).

In short, CI/CD could be understood as a constant flow of testing and deploying code: given the continuous and rapid nature of this practice, vast amounts of test log data are generated in short time spans.

Two major differences between a CI/CD-based environment and a more traditional software engineering setting are that of the testing and integration pace and that of the amount of debugging to be carried out after the testing finishes. When applying CI/CD, the pace of the testing and integration to the mainline is high (a fact that comes with constant production of vast amounts of log data), but the amount of debugging after testing is lower, given that software is being incrementally validated in short time frames. When applying a more traditional workflow, testing and integration take place less frequently, a fact that can come with the disadvantage of the developers facing a mainline that is so different from their baselines that the time needed for integrating exceeds the time it took to apply new changes (a situation colloquially referred to as “integration hell”) [51, 15].

3.2 Software encapsulation via containers

A good practice concerning software design and development for medium- to large-sized projects is to encapsulate the functionality of the software in containers [52].

When an operating system (OS) is containerized, it allows for the existence of multiple isolated environments called containers [53, 54]. Each container engine runs independently on top of the host OS of a given machine, as can be visualized in Figure 6. In a conceptually layered scheme, each container's libraries are placed separately on top of each container engine, over which the actual applications are located. This general framework can be seen in Figure 6a. Given that each container engine shares the same host OS, this practice constitutes a lightweight form of OS-level virtualization [52].

Not only does containerization provide container independence and hence security, but it also allows for managing container memory in an independent fashion with respect to other containers. Additionally, the containerized software is shipped so that it is not OS-dependent, meaning that the final software is transparent to the final host OS: the developers do not need to take it into account when undertaking the development. Depending on the implementation, containers can be split among different servers, as can be appreciated in Figure 6b.

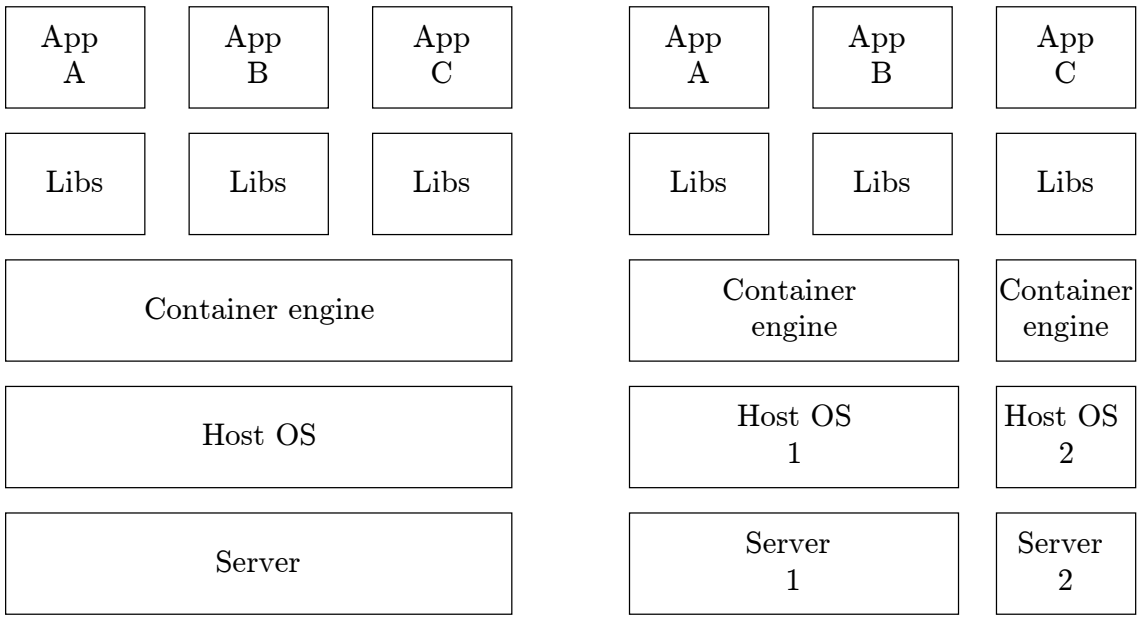
Containers are lightweight and stand-alone, as they include all necessary parts by themselves, that is, the produced code, runtime, system tools, libraries, and settings [54]. Overall, a container could be defined as a standardized, encapsulated, and self-reliant run-time environment based on an OS.

Designing and developing software in a containerized manner provides the benefit of establishing independent units in the developed system, thus creating boundaries in the behavior of the software program: on the one hand, from the software development perspective, this allows for separating the structure of the code and providing all the benefits that have been mentioned throughout this section; on the other hand, from the viewpoint of software testing, containers establish a unit in the behavior of the code, clearly separating the elements to be tested in unit and integration testing. The latter aspect is dealt with in Section 3.3.

3.3 Software testing: levels, suites, and regression testing

Once developed, software needs to be validated to ensure it works flawlessly, a task accomplished by means of testing [7].

Tests are automated executions of a program or its parts with a given set of inputs that lead to a comparison between the obtained results in each run and their expected outcomes. Naturally, the total number of test cases that form the test plan has to be finite, requiring a suitable selection from all available test conceptualizations.



(a) Containerization architecture on a single server.

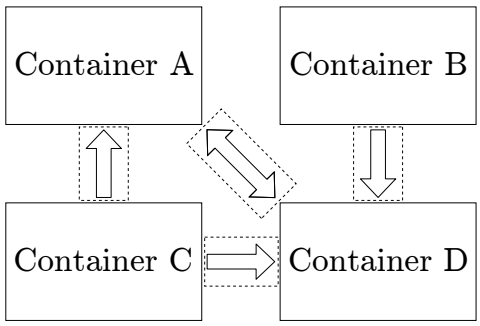
(b) Containerization architecture on multiple servers.

Figure 6: Containerized deployments for single and multiple servers [53, 54].

Software tests are categorized in a hierarchy according to their scope [55]. Firstly, unit testing validates a program in terms of its smallest separable elements, so that these are tested in an independent fashion with respect to one another. In a containerized environment, unit tests can be designed to tackle all the parts of a single container. The next level corresponds to integration testing, where the interactions between different units are validated (containers in a containerized setting). Lastly, the highest level corresponds to system testing, where the whole program in its entirety is validated. The scope of each test level can be further visualized in Figure 7, where Figures 7a, 7b, and 7c correspond to unit, integration, and system testing, respectively.

Besides their conceptualization in levels, tests are grouped in suites, which are collections of test cases in terms of the similarity of their evaluated functionality, as can be seen in Figure 8 [56, 55].

Additionally, when new modifications take place in the developed program by introducing or changing code, regression tests are run to ensure that no unintended effects have been caused in the form of bug introduction: they retest earlier passed tests in a suite to guarantee that the previously established behavior remains unchanged after incremental code updates [55].



(a) Scope of unit testing.

(b) Scope of integration testing.

(c) Scope of system testing.

Figure 7: Testing levels in a containerized software design framework [55]: containers (units) interact with each other, as displayed with the arrows. The dotted lines enclose the scope of a given test level.

3.4 Testing environment

This work is based on the software testing environment of a software development and testing team at Ericsson Finland: it follows the agile methodology via the XP framework, where the produced software is containerized, and CI/CD is applied. Its overall testing activity is displayed in Figure 9.

Once the developers have produced functional code, they commit it to a repository. This event triggers the CI tool, which starts executing unit and integration tests: these are run in dedicated servers, and they produce as outputs a binary result (either pass or fail) and a collection of test logs. Among others, the logs gather server analytics obtained during the evolution of the tests. Since the servers use the Hypertext Transfer Protocol (HTTP) as a means of communication, the responses are standardized according to their category. A three-digit code serves as an identifier, where the first numeral alludes to the class a response belongs to, as can be seen in Table 1 [57].

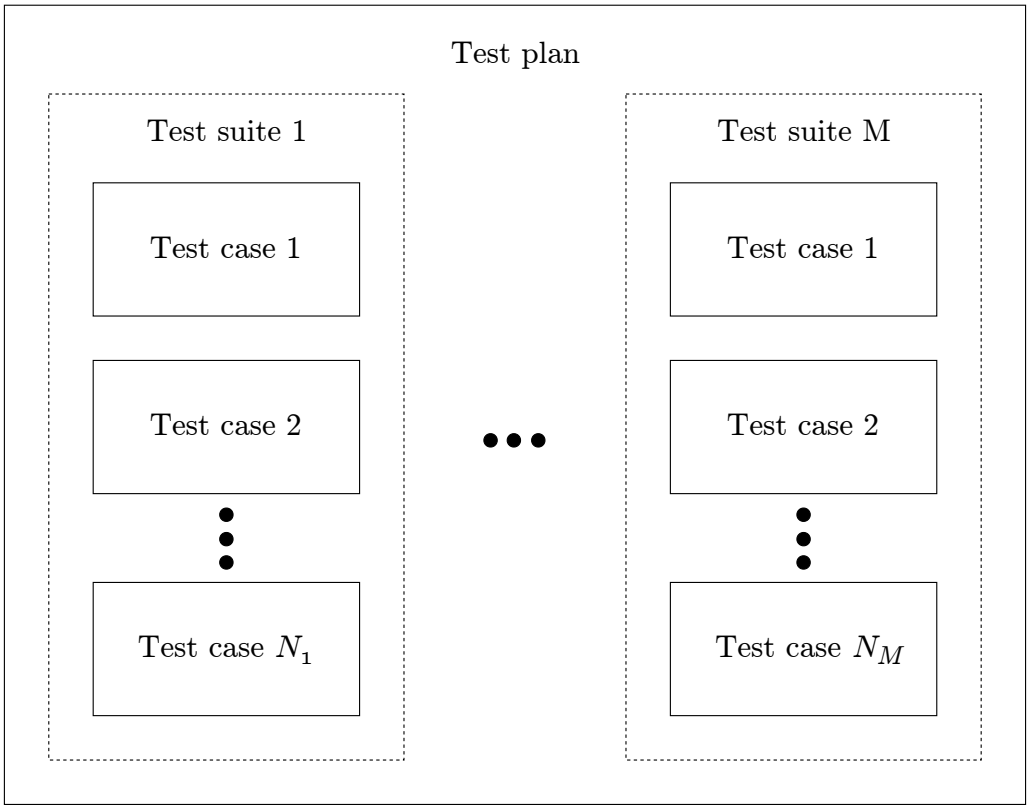


Figure 8: Hierarchical categorization of the test plan, test suites, and test cases: the test plan consists of suites, each of them gathering all cases that evaluate a similar functionality [56, 55].

In addition to the automated tests, a manual review of the code is carried out via a manual inspection tool: other developers are required to provide feedback for potential corrections.

If any issue is encountered at any of the two branches of validation (either software testing or manual inspection), the author of the code is notified: with the obtained feedback, the RCA stage can take place, where the author works on retrieving the source of the problem. Once that is achieved, the author can fine-tune the code, resulting in a new iteration of the cycle. When all tests are passed and the manual inspection provides positive feedback, the code is pushed to the main branch. This iterative process corresponds to the commit cycle that takes place every time new code is committed; it is illustrated in Figure 9a.

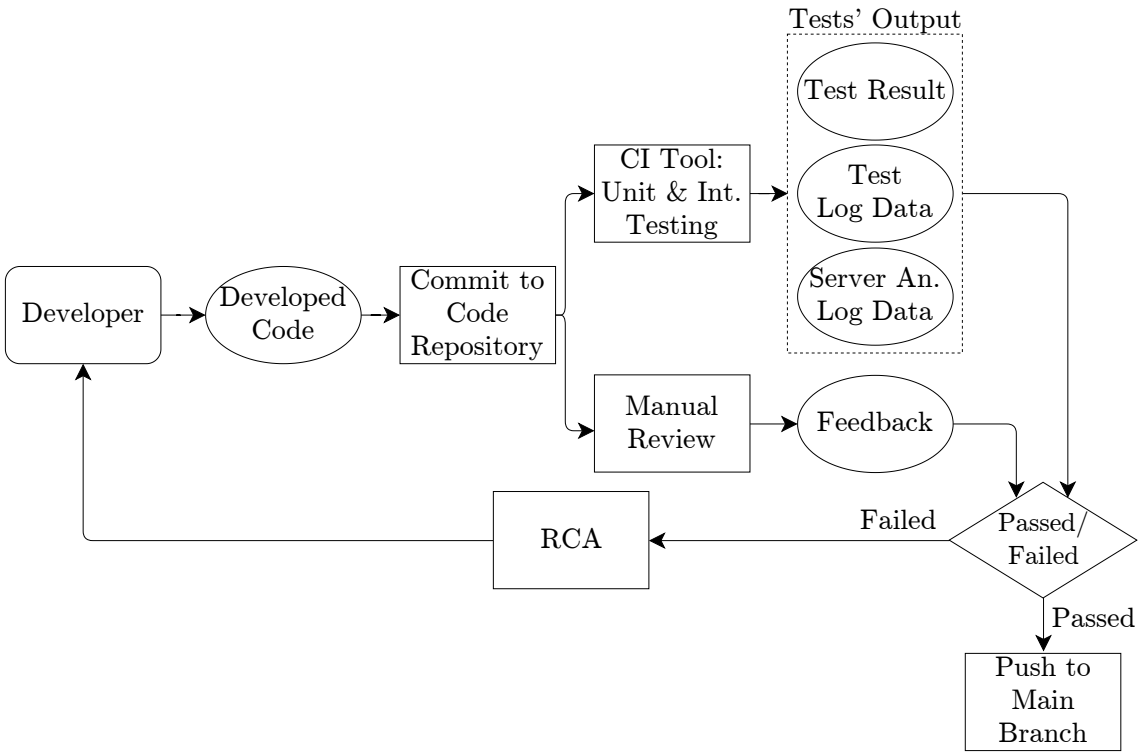
Furthermore, daily regression tests are continuously run, ensuring that the main branch contains functioning code and no new bugs have been introduced by pushing new updates. Their runtime of 2 hours translates into 12 regression tests per day. Lastly, nightly system tests are run to ensure that the software program in its entirety works flawlessly in the final main branch. These two periodic processes can be visualized in Figure 9b.

This research focuses on the obtained feedback regarding failed tests in the commit cycle and their subsequent RCA: the vast number of produced logs and the fast pace at which tests are carried out in the CI/CD environment motivates the automation of their analysis as to retrieve the root cause of the failed tests.

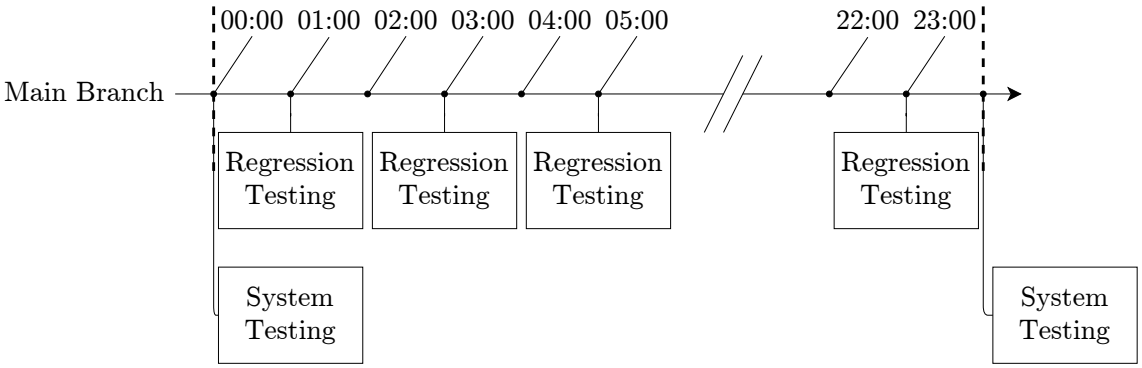
From all the produced log files, this thesis makes use of two distinct categories that are generated at each iteration of the commit cycle, from which the features are extracted (cf. Section 5.2). The first type gathers, among others, the test results for each of the run suites, as can be seen in the sample file displayed in Listing 1. The second type records each operation the containers perform as the tests are being run; whereas all of them are of interest in this work, special attention is put into four of them: server requests (which obtain an HTTP response from the contacted server), encountered errors, tracebacks that route back to a raised exception in the stack trace, and raised warnings. Their syntax can be appreciated in Listing 2.

HTTP status codes	**Category**	**Description**
1xx	Informational	The request was received, continuing process
2xx	Successful	The request was successfully received, understood, and accepted
3xx	Redirection	Further action needs to be taken in order to complete the request
4xx	Client Error	The request contains bad syntax or cannot be fulfilled
5xx	Server Error	The server failed to fulfill an apparently valid request

Table 1: HTTP status codes, as defined by the Internet Engineering Task Force [57].



(a) Commit cycle.



(b) Daily regression and nightly system testing.

Figure 9: Software testing environment of the agile team at Ericsson Finland this Master's thesis work is based on.

```
# Startup and configuration
...
START_TIME='YY-MM-DD HH:MM:SS.sss'
=========================================================
suite_1                                                 | PASS |
=========================================================
suite_1 :: test_1                                       | PASS |
1 test total, 1 passed, 0 failed
=========================================================
suite_2                                                 | PASS |
=========================================================
suite_2 :: test_1                                       | PASS |
---------------------------------------------------------
suite_2 :: test_2                                       | PASS |
---------------------------------------------------------
suite_2 :: test_3                                       | PASS |
3 tests total, 3 passed, 0 failed
# Additional suites
...
=========================================================
suite_N                                                 | FAIL |
=========================================================
suite_N :: test_1                                       | PASS |
---------------------------------------------------------
suite_N :: test_2                                       | FAIL |
---------------------------------------------------------
suite_N :: test_3                                       | FAIL |
---------------------------------------------------------
suite_N :: test_4                                       | PASS |
---------------------------------------------------------
suite_N :: test_5                                       | FAIL |
5 tests total, 2 passed, 3 failed
=========================================================
Tests                                                   | FAIL |
115 tests total, 110 passed, 5 failed
END_TIME='YY-MM-DD HH:MM:SS.sss'
# Resource cleanup
...
```

Listing 1: Sample log file containing granular pass/fail results per suite.

```
# Generic syntax
YYYY-MM-DD::HH:MM:SS.sss  container_1: ‘statement‘ ‘information‘
...
YYYY-MM-DD::HH:MM:SS.sss  container_2: ACTION ‘url‘ (HTTP/A.B CDE)
    => ‘results‘
YYYY-MM-DD::HH:MM:SS.sss  container_3: ERROR ‘description‘
YYYY-MM-DD::HH:MM:SS.sss  container_4: Traceback (most recent call
    last): ‘path‘/‘file‘ ‘line‘ ‘function‘ ‘exception‘
YYYY-MM-DD::HH:MM:SS.sss  container_5: WARNING REPORT ====
    DD-MM-YYYY::HH:MM:SS === ‘report‘
```

Listing 2: Sample log file recording the container activity (further specifying the four types of statements of particular interest).

4 Artificial intelligence, machine learning, representation learning, and deep learning

The field of AI focuses on the development of machines that demonstrate intelligence [58], i.e., the study of intelligent agents: automata that are able to perform intelligent tasks, mimicking human cognitive functions such as learning to take actions that maximize the chance of successfully achieving a goal in a specific environment [58, 59]. In this context, learning is understood as an entity's ability to progressively improve its performance on a specific activity [58, 19].

The scope of AI is itself not clearly defined and is changing as time passes. Since the conception of the first modern computer by Alan Turing [60], machines have demonstrated the ability to carry out an increasing number of human tasks, a fact that has led the general public to stop considering such abilities as "intelligent" [61].

Hence, at a specific moment in time, research in AI targets human abilities that have not yet been demonstrated by computers [62]. Once achieved, they are generally not acknowledged anymore as intelligent, a fact that is attested through Tesler's Theorem [61]:

> "AI is whatever has not been done yet".

Therefore, the discipline of AI puts its efforts into improving the overall performance of machines by making them undertake more complicated pieces of work. As an end goal, AI research pursues the development of artificial general intelligence, so that a machine could successfully carry out any intellectual activity that a human being can [63, 64].

AI is a multidisciplinary field, gathering knowledge from mathematics, computer science, psychology, linguistics, and philosophy, among others [58]. It was initially founded on the grounds that human intelligence could be precisely described so that a machine would be able to simulate it [65]. Since the twenty-first century, AI has experienced a rapid growth in interest in both academia and the corporate world, caused, among other reasons, by the advances in computer power, by the growth of available datasets due to the generalized use of the Internet, and by the increase of its theoretical understanding [26, 23] (cf. Section 2).

The achievement of intelligence by an automaton can be instructed manually and stored in knowledge bases, however, the difficulty in formalizing instructions and a computer's lack of flexibility when being hard-coded suggests the possibility of it learning by itself [23]. This capability is known as ML, which is the subfield of AI that focuses on providing automata with the ability to learn without being explicitly programmed to do so [66, 67].

ML can be itself understood as the mathematical tool that enables machines to learn a structure from data. Depending on the process and the available information a computer is provided with, in a traditional and plain categorization, its activity

can be subdivided into supervised and unsupervised learning [68, 69].

In a general and simplified fashion, a machine starts with a particular set of data, known as the training set [69, 70]. The machine tries to learn a structure from this set, and its action is corrected and directed towards improvement as time passes [19]. Once it has achieved a satisfactory performance, the machine is expected to generalize this structure to other unknown datasets that share similarities with the training set.

From a collection of features, which are data points from the problem in question, a machine makes predictions [69]. These predictions are based on a set of parameters that are adjusted by the machine, steering its performance towards improvement. Depending on the availability of ground-truth values for the predictions, the problem is of supervised or unsupervised learning, respectively. For the former case, these values, known as labels, can be used as a way of quantifying the goodness of the machine's performance. For the latter scenario, different data is made use of for this purpose (like the reconstruction error of the output with respect to the input in the case of an autoencoder (AE), as is seen later). The function that measures this is called the loss function, and it maps a specific cost to the performance, where the machine's objective is to minimize it and either get its predictions closer to the ground-truth values or enhance them in the way the loss function has been defined [69, 71]. Supervised algorithms can be further sub-categorized regarding the essence of the pursued output: regression for continuous results (e.g., the temperature in a city at a given moment in time), and classification for discrete values (e.g., whether a picture displays a child or an adult person) [69].

Generally, as has been previously hinted, a machine does not act on the raw data itself, but on a more compact representation of it, expressed through a collection of features [23, 72]. These features are not generally influenced by the system itself, making its selection one of the most critical steps to be carried out in the design of an ML solution. The sub-branch of ML that deals with the discovery of new data representations from the original features is known as representation learning (RL).

In certain scenarios, an RL system pre-processes the data in such a way that it can be handled afterward with an ML algorithm. A prime example of such a technology is the AE, a system comprised of a cascaded encoder and decoder function, where the encoding is constrained, and the subsequent decoding preserves the original structure of the data as much as possible. The basic architecture of an AE is shown in Figure 10.

However good properties RL offers, only relatively simple structures can be discovered from the data: if one wishes to separate different factors of variation, RL is generally not good enough [23]. Factors of variation can be understood as human constructs that help to describe attributes of observed data, and they generally do not manifest themselves in a directly observable fashion. As an example, the age, gender, geographical accent, and cadence of a speaker in a speech recording are factors of variation that help humans distinguish its author.

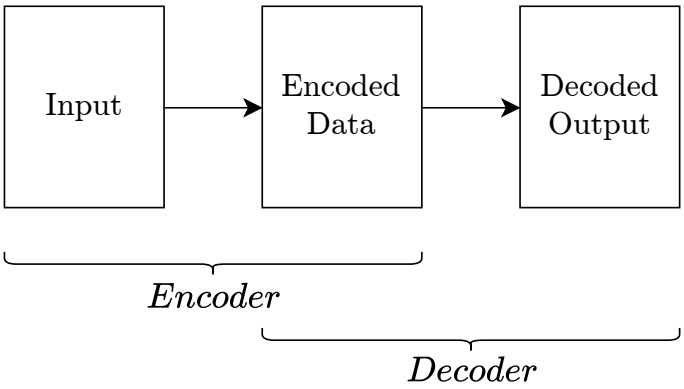


Figure 10: Basic structure of an AE, mapping an input to an encoded representation through the encoder, and from there to the decoded output via the decoder.

Extracting factors of variation from raw data is generally a hard task, but it can be tackled employing DL, the sub-branch of RL that obtains several representations in terms of other, simpler representations. This leads to a layered conceptual scheme, where more complex structures are obtained from simpler structures as the layers get "deeper".

From an analytic perspective, a DL system constructs a complex mathematical function from the cascading of simpler functions. The entities that compute these functions and therefore perform DL are called artificial neural networks (ANNs), a name inherited from their inspiration by biological neural networks in animals' brains [70].

An example of an ANN is the multilayer perceptron (MLP) or feedforward ANN, which performs a collection of cascaded function compositions. Its action can be represented as a graph, as shown in Figure 11. Each function composition operation is obtained by means of "neurons", which are grouped in layers according to their "depth". Normally, an ANN is formed by an input layer, a variable number of hidden layers, and an output layer: the input layer has a number of neurons equal to the number of features in the dataset, the hidden layers then obtain cascaded data representations, and the output layer stores the final values.

The previously tackled AE can be itself considered an ANN that is shallow, given that it only has one hidden layer. Nonetheless, adding more hidden layers to its architecture would enable for its action to share properties of both RL and DL.

The categorization this section has tackled can be further visualized in the Venn diagram displayed in Figure 12.

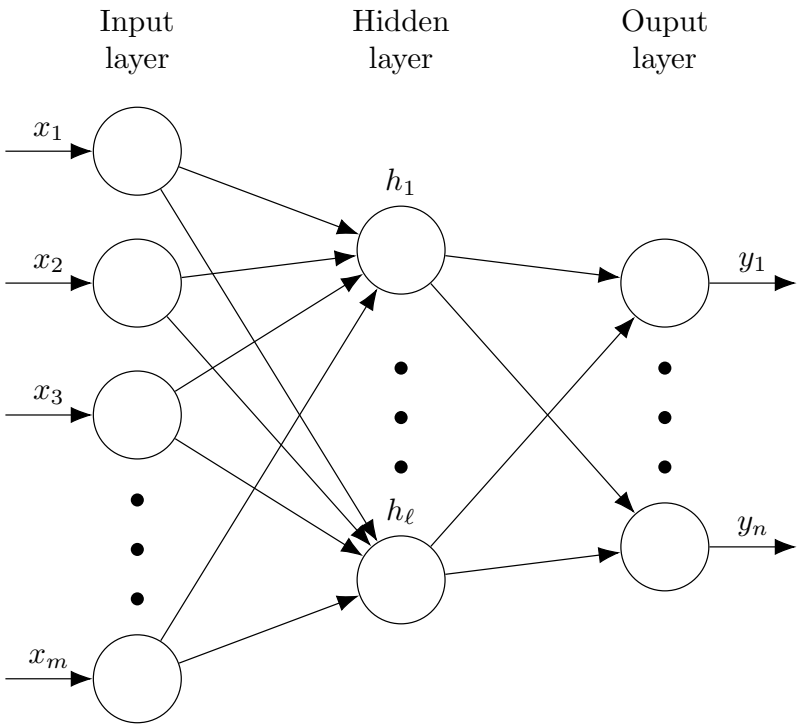


Figure 11: Densely connected MLP architecture, with m input neurons, ℓ neurons in the hidden layer, and n output neurons. The number of hidden layers determines the depth of the ANN.

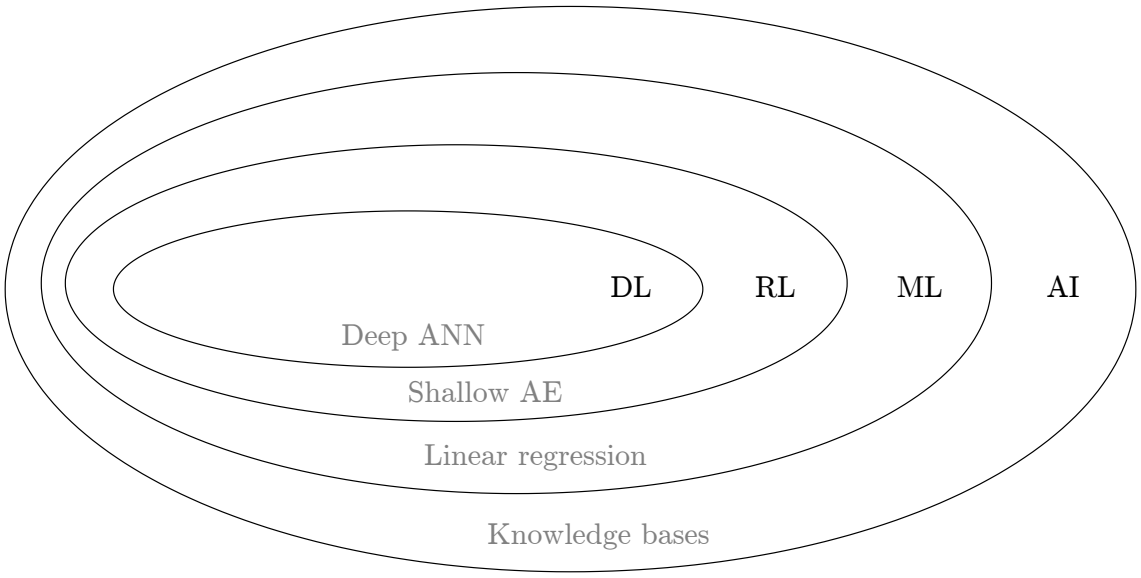


Figure 12: Venn diagram showing the relationship of AI and its sub-branches, along with an example for each field. Figure adapted from [23].

4.1 Learning process of an ML system

There are a number of elements that constitute an ML solution and describe its learning process [69]. For the sake of simplicity, the focus of this subsection is put in a supervised learning problem where the output is continuous (i.e., regression), whose derivation is similar to that of an unsupervised learning scenario.

A machine starts having some knowledge about a set of ground-truth values through their labels, stored in vector $\boldsymbol{y}$, and their features, gathered in matrix $\boldsymbol{X}$, where each row corresponds to a given data point, and each column represents a feature. With those, a certain kind of prediction or hypothesis function $h(\boldsymbol{X})$ is aimed so that it generalizes to new data in a satisfactory fashion. The hypothesis function relates to the features in a certain way, bounded by the problem's specification. In the case of linear regression, the relationship is linear:

$$h(\boldsymbol{X}) = \boldsymbol{X}^T \boldsymbol{w}, \tag{1}$$

where $\boldsymbol{w}$ is a weight vector, being responsible for fine-tuning the prediction by controlling the contribution of each feature. Analytically, the pursued hypothesis function $h(\boldsymbol{X})$ belongs to a given hypothesis space $\mathcal{H}$, containing all feasible and pursued mappings of a given kind.

Given the knowledge of the labels, a cost function can be defined so that it minimizes the average squared difference between the ground-truth values and their predictions:

$$J((\boldsymbol{X}, \boldsymbol{y}), h) = \frac{\sum_{i=1}^{N} (y_i - h(\boldsymbol{X}_{i,:}))^2}{N}, \tag{2}$$

where N represents the number of examples.

This particular cost function is known as the mean squared error (MSE) loss function, and the ML system in question iterates the search of weights until it finds a combination of values that minimizes its cost.

Gradient descent (GD) is an available method for accomplishing this [73, 69]. It iteratively updates the weight values following the direction of maximum steepness (gradient) of the cost function:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \alpha \nabla J\left(\boldsymbol{w}^{(t)}\right), \tag{3}$$

where t indexes the training iterations, α is the learning rate, and $\nabla J(\boldsymbol{w})$ is the gradient of the cost function.

High values of the learning rate ensure accelerating the convergence of GD, but risk overshooting the function's minimum, and in the worst-case scenario, never

converging. Alternatively, low values cause GD to converge at a slower pace, taking more iterations. Figure 13 shows an instance of a satisfactory evolution of GD for a bivariate convex cost function.

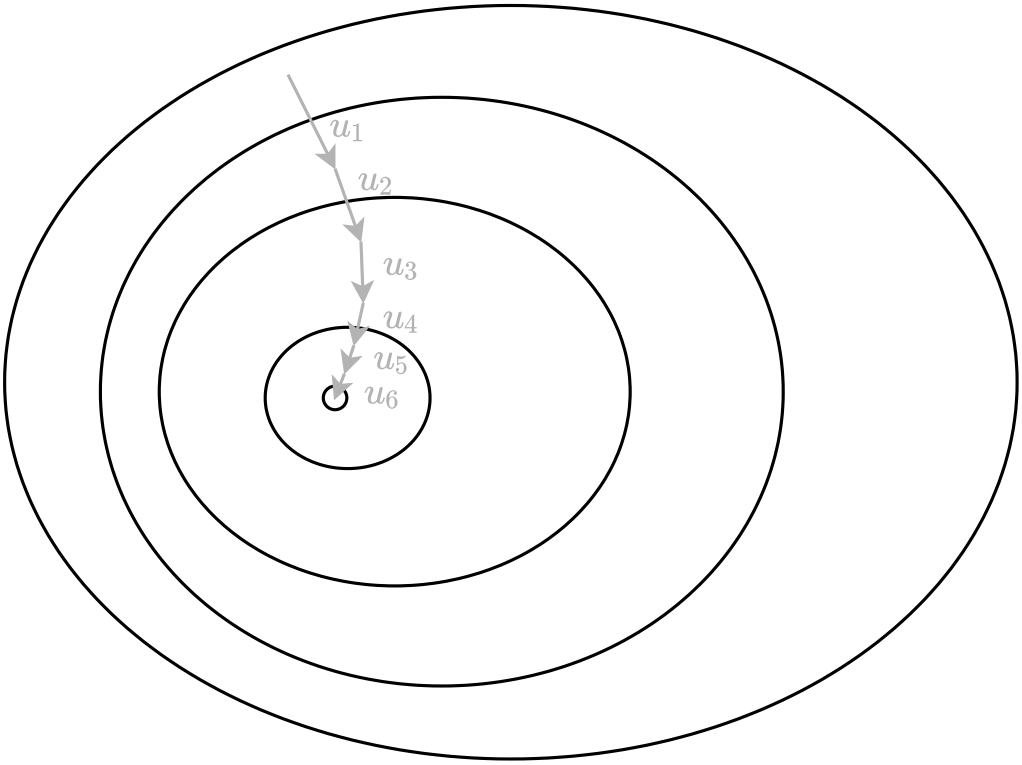


Figure 13: GD example: six subsequent updates over a bivariate convex cost function, depicted as a series of level sets, leading to the minimization of the cost function.

While GD performs well (provided that the learning rate is set to a working value), it computes the gradient of the cost function for all the training examples, a fact that might be a disadvantage over large training sets, making its evolution usually slow.

Stochastic GD (SGD) is a variation of GD that updates the parameters for each training example [73, 69]:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \alpha \nabla J\left(\boldsymbol{w}^{(t)}, (\boldsymbol{X}_{i,:}, y_i)\right). \tag{4}$$

Its increased speed comes at the cost of complicating the convergence to an exact minimum value due to generally overshooting it. Additionally, SGD updates do not follow the path of overall maximum steepness, defining a curved path along the loss function instead.

As a toy model, Figure 14 shows three different predictions obtained with three different weight vectors, where the aim is set at finding the best linear prediction on the training set. Afterward, the prediction is expected to generalize well to new data.

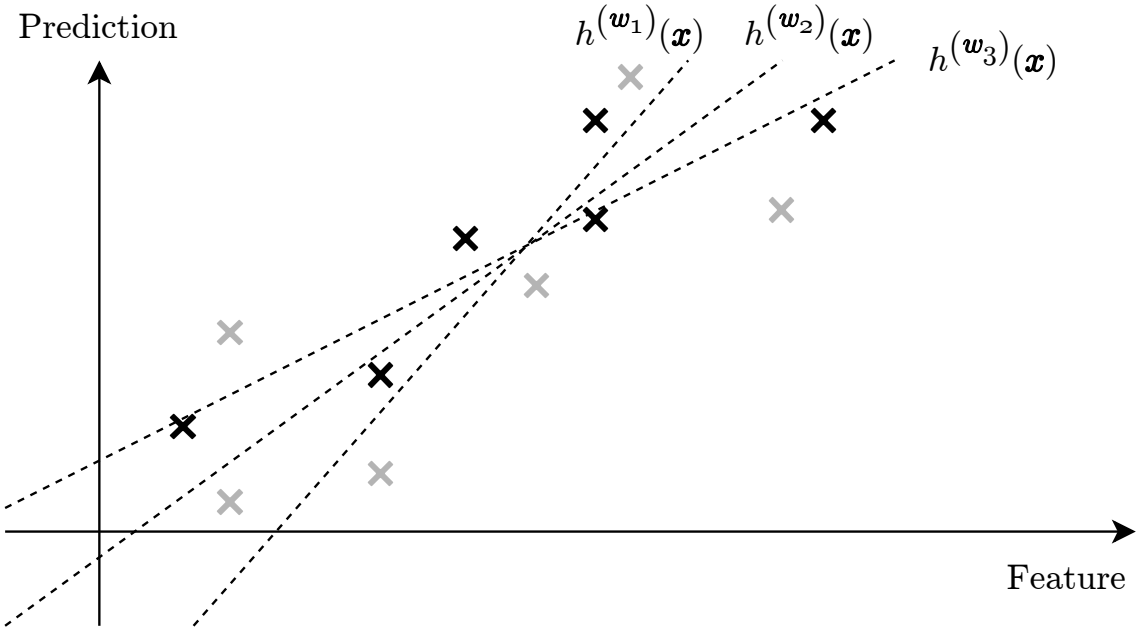


Figure 14: Predictions from an ML system based on some data in a univariate feature space. The machine trains its model to fit the training data (black crosses) by minimizing the cost function that is dependent on the weights $\boldsymbol{w}$. Given that this is an iterative process, several predictions are generated (being three shown in the picture). Once the ML system achieves the prediction with the lowest associated training loss, it expects to generalize its action to other previously unseen data (grey crosses). Figure adapted from [69].

Two obstacles are present during this process [23, 19, 69]. On the one hand, if the prediction fits the training set too accurately, it might not generalize well, a case known as overfitting. On the other hand, if the prediction returns a high error value on the training set itself, underfitting takes place, where the predictor fails at the initial stage. Whereas the latter can be tackled by working on minimizing the cost function's value on the training set, the former can be dealt with by means of regularization, understood as an action taken on the cost function to reduce its generalization error without affecting its training error. For this purpose, an extra additive term can be added to the cost function:

$$J\left(\left(\boldsymbol{X},\boldsymbol{y}\right),h\right)=\frac{\sum_{i=1}^{N}\left(y_i-h\left(\boldsymbol{X}_{i,:}\right)\right)^2}{N}+\lambda R\left(J\right). \tag{5}$$

The regularization function $R\left(J\right)$ smoothens the final prediction: it adds an extra penalization to the cost function, usually imposed on the complexity of J, forcing the learning process to achieve a simpler result on the training set. The amount of regularization to be applied is controlled by the regularization parameter λ.

Two of the most well-known regularization functions are used in this thesis, namely, L1 and L2 regularization [23].

L1 regularization introduces a penalty in the form of the sum of the absolute values of the weight vector elements:

$$R(J) = \|\boldsymbol{w}\|_1. \tag{6}$$

On the other hand, L2 regularization, which also goes by the name of ridge regression, takes into account the sum of squared magnitudes of the weights as a penalty term, thus penalizing large weight values with a greater impact than L1 regularization (for a shared value of the regularization parameter λ):

$$R(J) = \|\boldsymbol{w}\|_2. \tag{7}$$

In order to provide a convincing result on the training of a given ML algorithm, a simple and good practice consists of assessing its performance over a training and validation set [19, 69]: the training set is used for minimizing the cost function and fine-tuning the parameters, and the validation set ensures that no overfitting is taking place and that the performance generalizes well. Finally, once the optimal solution is found on both sets, a test set is used for reporting the final performance metrics, as it has not been used for enhancing the algorithm's performance [19, 23].

4.2 ANNs

Different ANN architectures are available for different purposes. This thesis puts its attention on AEs and MLPs, the former for pre-processing data that is later clustered, the latter corresponding to the feedforward ANN architecture used for classification.

The processing unit of an ANN receives the name of "neuron" [70]. A neuron consists of three essential elements: a set of connecting links that feed the neuron, an adder that sums the different input values, and an activation function that constrains the output.

The k^{th} neuron in a network gets a collection of values x_{k1} to x_{kp} corresponding to either input features if the neuron is located in the input layer, or otherwise, to values processed by another neuron. These inputs get multiplied by weights w_{k1} to w_{kp}, which control the influence of each input and thus tune the action of the neuron. Afterward, the processed inputs are added together. A fixed bias term b_k is included in this summation, whose action is to either increase or decrease the sum v_k:

$$v_k = \sum_{j=1}^{p} w_{kj} x_{kj} + b_k. \tag{8}$$

The final amplitude value is scaled to a given range, usually $[0, 1]$ or $[-1, 1]$. This is achieved by applying a nonlinear function $\varphi(\cdot)$ to v_k, known as the activation function:

$$y_k = \varphi(v_k). \tag{9}$$

The action a neuron performs can be represented in the form of a block diagram, as can be seen in Figure 15.

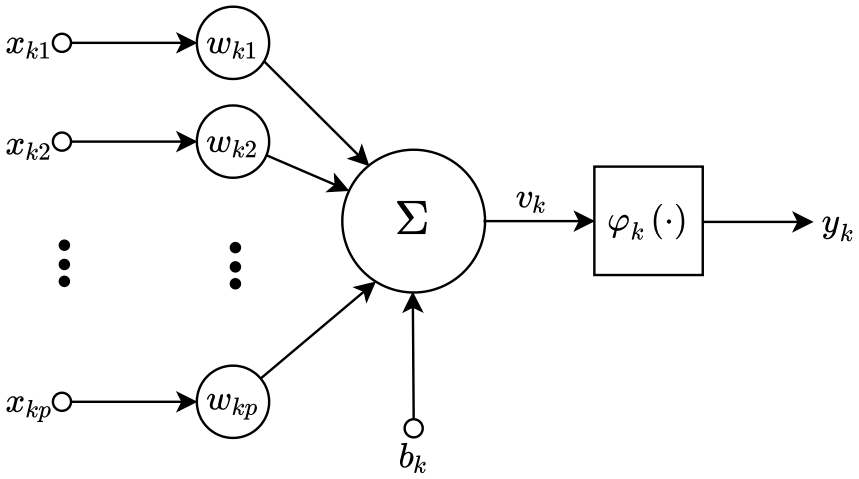


Figure 15: Block diagram of a neuron. Figure adapted from [70].

Different activation functions exist for applying a final constraint on the neurons' summation; this thesis focuses on the sigmoid, hyperbolic tangent (tanh), rectified linear unit (ReLU), and softmax activation functions, whose transfer functions are described next. Given that sigmoid, tanh, and ReLU are univariate functions (unlike softmax, which is a multivariate function), their plots are gathered in Figure 16.

The sigmoid activation function scales the values between 0 and 1 in such a way that input values around 0 undergo a linear transformation, whereas extreme values are scaled to 0 and 1 depending on them being negative or positive, respectively (cf. Figure 16a):

$$\varphi_{\text{sigmoid}}(v_k) = \frac{1}{1 + e^{-v_k}}. \tag{10}$$

Similarly, the tanh activation function squashes the input values in a range bounded by -1 and 1 (cf. Figure 16b):

$$\varphi_{\tanh}(v_k) = \frac{e^{v_k} - e^{-v_k}}{e^{v_k} + e^{-v_k}}. \tag{11}$$

ReLU is defined as a half-wave rectifier, which nullifies negative values and keeps positive values unaltered, not establishing an upper bound on the output values (cf. Figure 16c):

$$\varphi_{\text{ReLU}}\left(v_k\right) = \max\left(0, v_k\right). \tag{12}$$

As of the time of writing this thesis, ReLU is the most popular activation function in deep ANNs due to its demonstrated efficient performance in the training process [74, 75].

Lastly, the softmax function exponentially normalizes the values of a given layer to the $[0, 1]$ range so that all of them add up to 1:

$$\varphi_{\text{ softmax }}\left(v_k\right) = \frac{e^{v_k}}{\sum_{i=1}^{q} e^{v_i}}, \tag{13}$$

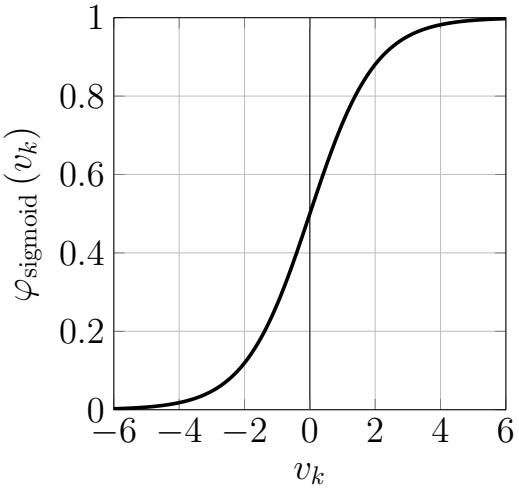
where q is the number of neurons in the current layer.

Due to its nature, the softmax activation function is generally used at the output layer in an MLP classifier: each output neuron stores a value that can be interpreted as the probability that a given example has of belonging to a specific category (cf. Section 4.3).

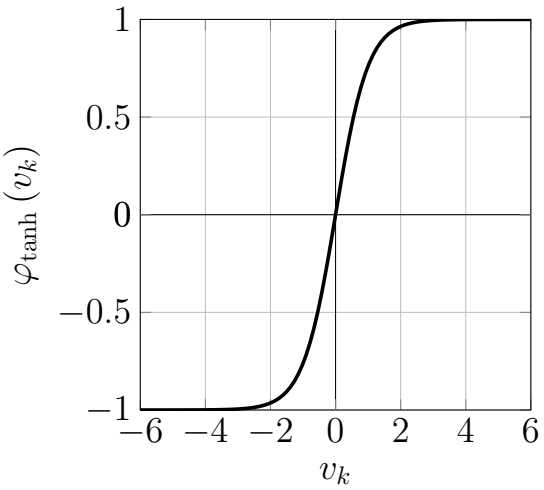
In DL, an ANN-specific GD variation known as the backpropagation algorithm is generally applied for training and optimizing weights [23]. For an input $\boldsymbol{X}$, the ANN computes the calculations through the neurons sequentially, until producing the final predicted values $\widehat{\boldsymbol{y}}$, a process known as forward propagation. This allows for the computation of the cost $J\left(\left(\boldsymbol{X}, \boldsymbol{y}\right), h\right)$. Afterward, backpropagation permits the information of the cost to flow backward so that its gradient is computed; the weights can then be optimized. This solution, initially mentioned in the context of DL by Werbos in 1975 [76], proves to be a more efficient computation than a direct analytic derivation of the gradient. In this context, the combination of a forward pass and backward pass of all training examples is given the name of "epoch", a unit that quantifies the amount of training an ANN undergoes.

Specific to ANN, the techniques of dropout and batch normalization can be used as regularizers [23]. Dropout consists of the deactivation of several neurons at random throughout the iterations of the training process. This makes it harder for the network to overfit: communication between all neurons is blocked at certain moments; thus, the ANN is less likely to learn a more established structure from the data. Batch normalization corresponds to standardizing the neurons' inputs in the hidden layers to be zero mean and unit variance, which results in a slight regularization effect due to the values of the inputs being bounded:

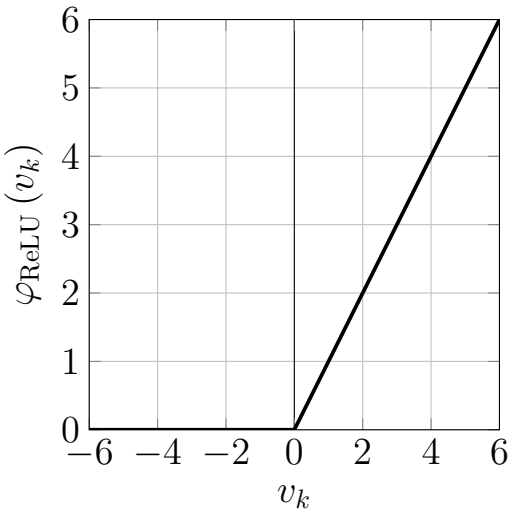
$$\widehat{\boldsymbol{x}}_k = \frac{\boldsymbol{x}_k - \mathrm{E}\left[\boldsymbol{x}_k\right]}{\sigma\left[\boldsymbol{x}_k\right]}. \tag{14}$$



(a) Sigmoid activation function.



(b) tanh activation function.



(c) ReLU activation function.

Figure 16: Transfer functions of the sigmoid, tanh, and ReLU activation functions.

A special consideration for DL concerns the optimization of the neurons' weights: even though SGD already speeds up the evolution of the training process, it can still take a considerable amount of time depending on the ANN architecture in question. In order to overcome this, several optimization algorithms have been developed, among which, this thesis focuses its attention on adaptive moment estimation (Adam), proposed by Kingma and Ba [77, 23, 73].

This algorithm computes adaptive learning rates for each parameter. It starts by calculating an exponentially weighted average of past gradients and their squares:

$$m_w{}^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\,\nabla J\left(\boldsymbol{w}^{(t)}\right), \tag{15}$$

$$v_w{}^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)\left(\nabla J\left(\boldsymbol{w}^{(t)}\right)\right)^2, \tag{16}$$

where m_w and v_w are the first and second moments of the weights, and β_1 and β_2 are hyperparameters to be tuned.

These values get then corrected as to counteract their intrinsic bias towards zero (due to the fact that $m_w{}^{(t=0)}$ and $v_w{}^{(t=0)}$ are initialized as vectors of zeros):

$$\widehat{m}_w = \frac{m_w{}^{(t+1)}}{1-(\beta_1)^{t+1}}, \tag{17}$$

$$\widehat{v}_w = \frac{v_w{}^{(t+1)}}{1-(\beta_2)^{t+1}}. \tag{18}$$

The weights get then updated using both moments:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} - \alpha \frac{\widehat{m}_w}{\sqrt{\widehat{v}_w}+\varepsilon}, \tag{19}$$

where ε functions as a denominator's addend, storing a small value to avoid a division by zero.

4.3 Classification

Classification tasks constitute a branch of supervised learning that aims at predicting the category within k predefined candidates a given input belongs to [23].

This thesis focuses on using MLPs for this purpose: these ANNs are universal function approximators, where their good performance in regression analysis can be extrapolated to the categorical case of classification [78, 23]. Their general architecture consists of an input layer with as many neurons as features, an output layer with as many neurons as classes, and a given number of hidden layers, each with a certain number of neurons.

Each of the output neurons is assigned to a conceptual class. For the sake of consistency, the labels associated with the data need to match this assignment. When using a softmax activation function in the output layer, the final output values are interpreted as the probability a given example has of belonging to a specific class.

Thus, for each example, the class associated with the output neuron that stores the highest value is selected as the predicted category. The objective of the ANN is to adjust the weights of the neurons so that the classification maximizes the matching between the predicted and the ground-truth categories.

In this scenario, where the output layer of the MLP stores values that represent the probability of the input belonging to a category, the categorical cross-entropy loss function is used [79]:

$$J\left(\left(\boldsymbol{X}, \boldsymbol{y}\right), h\right) = -\sum_{i=1}^{N}\sum_{g=1}^{k} \boldsymbol{Y}_{i,g} \log\left(\widehat{y}_{i,g}\right), \tag{20}$$

where i indexes the examples, g indexes the classes, $\boldsymbol{Y}$ is the label vector converted to a one-hot-encoded matrix, and $\widehat{\boldsymbol{y}}$ is the one-hot-encoded vector of the final predictions:

$$\widehat{\boldsymbol{y}} = h\left(\boldsymbol{X}\right). \tag{21}$$

A one-hot-encoded matrix is a matrix with as many rows as examples and as many columns as the number of ground-truth categories: it is set to zero but for the indices whose columns match the ground-truth categories, where the value is equal to 1.

Minimizing the categorical cross-entropy causes the one-hot-encoded predictions to be closer to the labels.

4.4 Clustering analysis

Clustering analysis is a form of unsupervised learning where the task is set at grouping data points according to some similarity measure. It is usually used in the process of discovering a structure in unlabeled data [69].

Clustering can be further subdivided into hard and soft clustering. The former encompasses algorithms that assign one and only one cluster to a given example, whereas the latter takes a probabilistic approach, where an example is given a probability value of being assigned to a cluster.

Several proposals are available in each subgroup, among which k-means and Gaussian Mixture Models (GMMs) are of interest in this work, the former being an example of hard clustering, the latter an instance of soft clustering; these algorithms correspond to prime examples of their corresponding categories [69].

The k-means algorithm relies on an iterative approach that consists of two steps.

Initially, the number of clusters k is decided. The $(k \times m)$-dimensional cluster-center matrix $\boldsymbol{C}$ is then initialized at random, where m is the number of features.

In the first step, an assignment takes place, where each data point $\boldsymbol{X}_{i,:}$ is assigned to its closest cluster center, defined in terms of the Euclidean distance:

$$a_i = \underset{j \in \{1,\ldots,k\}}{\arg\min} \left\| \boldsymbol{X}_{i,:} - \boldsymbol{C}_{j,:} \right\|, \tag{22}$$

where $\boldsymbol{a}$ is a vector storing the assigned cluster for each example.

Then, each of the k cluster centroids gets updated as the average value of all data points assigned to that cluster:

$$\boldsymbol{C}_{i,:} = \frac{1}{|\{i : \boldsymbol{X}_{i,;} \in a_i\}|} \sum_{i:\boldsymbol{X}_{i,;} \in a_i} \boldsymbol{X}_{i,:}. \tag{23}$$

The algorithm ends once it has converged, a state that is usually defined as the difference in assignments in a number of consecutive updates lying below a particular threshold.

The simplicity of k-means comes at the cost of achieving rather simple linear boundaries, a situation that GMMs overcome by allowing more complex divisions in the m-dimensional feature space.

GMMs turn the problem of clustering into that of parameter estimation: the observed data points are now seen as realizations of a random vector modeled via a Gaussian probability distribution.

The probability distribution of the random vector thus depends on the cluster indices $\boldsymbol{c} \in \{1, \ldots, k\}$, with their respective associated mean vectors $\boldsymbol{\mu}$ and covariance matrices $\boldsymbol{\Sigma}$:

$$\mathrm{P}\left(\boldsymbol{X}_{i,:} | c_i\right) = \mathcal{N}\left(\boldsymbol{\mu}_{(c_i)}, \boldsymbol{\Sigma}_{(c_i)}\right), \tag{24}$$

where $\boldsymbol{\mu}_{(c_i)}$ and $\boldsymbol{\Sigma}_{(c_i)}$ correspond to the unknown mean vector and the unknown covariance matrix associated with the cluster indexed as c_i.

Since the cluster centers are unknown, they can be conceptualized as independent and identically distributed random variables (RVs) distributed over the set of k cluster indices. This leads to the model being a mixture of Gaussians, given that the marginal distribution is a superposition of Gaussian distributions:

$$\begin{aligned} \mathrm{P}\left(\boldsymbol{X}_{i,:}\right) &= \sum_{j=1}^{k} \mathcal{N}\left(\boldsymbol{\mu}_{(j)}, \boldsymbol{\Sigma}_{(j)}\right) \mathrm{P}\left(c_i = j\right) \\ &= \sum_{j=1}^{k} \mathcal{N}\left(\boldsymbol{\mu}_{(j)}, \boldsymbol{\Sigma}_{(j)}\right) p_j, \end{aligned} \tag{25}$$

where the parameters to be estimated are the underlying cluster distributions p_j, as well as the means $\boldsymbol{\mu}_{(j)}$ and covariances $\boldsymbol{\Sigma}_{(j)}$ of the normal distribution.

GMMs rely on the expectation-maximization algorithm for this purpose, whose approach shares certain similarities with the k-means algorithm: it consists of two iterative steps, namely, cluster assignment and parameter estimation update.

After the number of clusters k has been selected, an initial guess for $\widehat{p}_j$, $\widehat{\boldsymbol{\mu}}_{(j)}$, and $\widehat{\boldsymbol{\Sigma}}_{(j)}$ is carried out (being these the estimates of p_j, $\boldsymbol{\mu}_{(j)}$, and $\boldsymbol{\Sigma}_{(j)}$, respectively).

Then, the assignment of probabilities of each data point belonging to a cluster is undertaken:

$$A_{i,j} = \frac{\widehat{p}_j \, \mathcal{N}\left(\boldsymbol{X}_{i,:}; \widehat{\boldsymbol{\mu}}_j, \widehat{\boldsymbol{\Sigma}}_j\right)}{\sum_{q=1}^{k} \widehat{p}_q \, \mathcal{N}\left(\boldsymbol{X}_{i,:}; \widehat{\boldsymbol{\mu}}_q, \widehat{\boldsymbol{\Sigma}}_q\right)}. \tag{26}$$

After this is achieved, the estimates of the GMM parameters are updated:

$$\widehat{\boldsymbol{\mu}}_j = \frac{1}{\sum_{i=1}^{N} A_{i,j}} \sum_{i=1}^{N} A_{i,j} \boldsymbol{X}_{i,:}, \tag{27}$$

$$\widehat{\boldsymbol{\Sigma}}_j = \frac{1}{\sum_{i=1}^{N} A_{i,j}} \sum_{i=1}^{N} A_{i,j} \left(\boldsymbol{X}_{i,:} - \widehat{\boldsymbol{\mu}}_j\right) \left(\boldsymbol{X}_{i,:} - \widehat{\boldsymbol{\mu}}_j\right)^T. \tag{28}$$

As with k-means, the convergence is defined in terms of the amount of change in cluster assignments between updates lying below a given threshold.

The resulting m-dimensional regions follow smoother contours, which sometimes achieve more satisfactory results than the regions with linear boundaries obtained with the k-means algorithm.

For both algorithms, the membership of new data can be retrieved, meaning that the cluster boundaries can be defined on a training set, and later be used for assigning clusters to validation and test examples. When doing so, in order to keep all output cluster labels consistent, certain matching algorithms can be used, among which, the Hungarian algorithm is of interest in this work [80]. After having obtained the confusion matrix $\boldsymbol{C}$, which displays the fixed clusters as columns and the predicted clusters for a given set as rows (storing the counts of matches), it aims at minimizing the cost of assigning a row to a column:

$$a_j = \arg\min_j \sum_i \sum_j \boldsymbol{C}_{i,j} \boldsymbol{B}_{i,j}, \tag{29}$$

where $\boldsymbol{B}$ is a Boolean matrix whose unit entries correspond to assigned rows and columns.

In order to evaluate the performance of clustering, different metrics have been proposed, which are introduced in Section 6.1.

4.4.1 Pre-processing techniques for clustering analysis

While clustering algorithms are usually fed with the original data features, RL can be used for obtaining new data representations that might help in the clustering task.

For instance, PCA is a statistical technique that pursues dimensionality reduction while retaining as much variation in the original data as possible [81, 69]. The original data undergoes an orthogonal transformation so that the possibly correlated initial features are converted to a collection of linearly uncorrelated variables that take the name of principal components.

Informally, the new representation of the data is aimed at being as concise as possible, while at the same time, at conveying as much of the original information as possible, in a way that the new variables are independent of one another.

For this sake, the covariance matrix of the features $\boldsymbol{\Sigma}$ undergoes eigenvalue decomposition:

$$\boldsymbol{\Sigma} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_D) \begin{pmatrix} \lambda_1 & \ldots & 0 \\ 0 & \ddots & 0 \\ 0 & \ldots & \lambda_D \end{pmatrix} (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_D)^T, \tag{30}$$

where $(\boldsymbol{u}_1, \ldots, \boldsymbol{u}_D)$ are the orthonormal eigenvectors, $(\lambda_1, \ldots, \lambda_D)$ are the decreasing eigenvalues, arranged in a diagonal matrix, and $\boldsymbol{\Sigma}$ is the covariance matrix:

$$\boldsymbol{\Sigma} = \boldsymbol{X}^T \boldsymbol{X}. \tag{31}$$

From the total number of principal components D, a number $d < D$ is selected to reduce the dimensionality of the data. The d extracted components are arranged as a compression matrix:

$$\boldsymbol{W}_{\text{PCA}} = (\boldsymbol{u}_1, \ldots, \boldsymbol{u}_d)^T. \tag{32}$$

The new principal components are then obtained as a linear product of the compression matrix and the original features:

$$\boldsymbol{X}_{\text{PCA}} = \boldsymbol{W}_{\text{PCA}} \boldsymbol{X}. \tag{33}$$

The percentage of retained variance from the original features can be calculated as:

$$\text{Retained Variance}\,(\%) = \frac{\sum_{j=1}^{d} \lambda_j}{\sum_{i=1}^{D} \lambda_i} \times 100, \tag{34}$$

a value that can be used for deciding the number of components to be retained d [82].

Whereas PCA achieves a new representation that maximizes data variation, and therefore intuitively might feed the clustering algorithm with an uncorrelated set of variables, AEs can generalize its action to nonlinear mappings that outperform it [83, 23]. This leads to the idea of feeding a clustering algorithm with the encoded representation obtained by an AE.

Nonetheless, a standard AE does not pursue achieving an encoded representation where the data is grouped in clusters. With this in mind, Song et al. [84] recently proposed combining both of the aforementioned actions (RL and clustering), designing a custom AE whose loss function incorporates the original reconstruction error defined in terms of the MSE, and the k-means Euclidean distance for cluster assignment:

$$J\left(\left(\boldsymbol{X}, \boldsymbol{c}\right), h\right) = \frac{1}{N}\sum_{i=1}^{N}\left\|\boldsymbol{X}_{i,:} - \boldsymbol{X}'_{i,:}\right\|^2 - \eta\sum_{i=1}^{N}\left\|f^{(t)}\left(\boldsymbol{X}_{i,:}\right) - \boldsymbol{c}^*_{(i)}\right\|^2. \tag{35}$$

The first term corresponds to the loss function of the AE, i.e., its reconstruction error: $\boldsymbol{X}$ is the feature matrix, and $\boldsymbol{X'}$ is the output matrix reconstructed by the decoder.

The second term is the Euclidean distance as taken from the k-means algorithm, whose contribution is controlled via the parameter η. $f^{(t)}\left(\cdot\right)$ is the encoder function at the t^{th} iteration, and $\boldsymbol{c}^*_{(i)}$ is the closest cluster center corresponding to the i^{th} element in the code layer:

$$\boldsymbol{c}^*_{(i)} = \underset{\boldsymbol{C}^{(t-1)}_{j,:}}{\arg\min}\left\|f^{(t)}\left(\boldsymbol{X}_{i,:}\right) - \boldsymbol{C}^{(t-1)}_{j,:}\right\|^2. \tag{36}$$

The AE first optimizes the encoding function $f\left(\cdot\right)$ and then updates the cluster centers:

$$\boldsymbol{C}^{(t)}_{j,:} = \frac{\sum_{\boldsymbol{X}_{i,:}\in\boldsymbol{C}^{(t-1)}_{j,:}} f^{(t)}\left(\boldsymbol{X}_{i,:}\right)}{\left|\boldsymbol{C}^{(t-1)}_{j,:}\right|}. \tag{37}$$

In summary, this modified version of the AE aims at finding a compromise between the reconstruction error and obtaining an encoded layer where the data is grouped in k clusters.

A special consideration that AEs must fulfill in order to learn the most salient features from a given input is that they have to be undercomplete, meaning that their hidden layer (or layers) must be of a lower dimension than their input layer [23].

5 Automating RCA in agile CI/CD software testing

5.1 Iterations of the development work

As presented in Section 1.3, the author of this thesis follows the research methodology of action research. In total, two major iterations took place during the development of this Master's thesis work.

5.1.1 First iteration: clustering of the unlabeled log data

In the first iteration, the author of the thesis assessed clustering algorithms on the log data to determine whether a structure could be found from it that would correlate to the root causes of the failed tests.

During its first phase of inquiry, the author defined the initial research questions, which correspond to the first and third final questions that are gathered in Section 1.3; the second question is not posed at this stage given that no ground-truth root cause categories were pursued in the first iteration. A literature review followed, where the author studied the previous attempts in clustering unlabeled log data. Afterward, the design process was carried out, where the initial short-term deadlines were set by the author and the thesis supervisors.

In the second phase of action, the author gained knowledge on the testing environment (described in Section 3.4) and determined which log files would be necessary to gather and process (also gathered in Section 3.4). In order to fetch them, the author wrote a log collection program. Subsequently, the author analyzed the log files manually and interviewed the testing engineers to understand what they paid attention to in the log files in order to retrieve the root causes of the test failures.

After having explored the debugging process of several testing engineers, one could see how the pass/fail ratios and their occurrence across suites seemed to correlate with the origin of the failures. Additionally, the responses that the servers provided to the containers' requests and the number of times the containers were invoked gave hints on why certain tests failed. These findings lead to a conceptual categorization of four feature groups that best summarize the evolution and result of a test, as assessed by the testing engineers (cf. Section 5.2). The features are obtained by processing the raw log files: the author wrote a software program for processing the raw log data, adapted to the environment this work is based on in such a way that it locates the statements in the log files that relate to the envisioned feature groups and extracts the features.

The author then tested the clustering algorithms presented in Section 4.4 with different cluster sizes: from all the tested values, the best results were achieved with $k = 5$ and $k = 10$ clusters. In the third phase of analysis, a qualitative assessment

of the obtained results was undertaken: even though good qualitative correlations were attained with several clusters and failure causes, the meaning of other clusters remained vague. As an example, two clusters when selecting $k = 10$ with the k-means algorithm seemed to correlate well with two different containers crashing on execution, causing the corresponding tests to fail. The results when making use of the k-means algorithm and GMMs can be visualized in Figures 17 and 18, respectively; the chosen visualization plots the failed tests' normalized invocations of a specific container with respect to the overall test success ratio. This visualization displays a mild correlation between the number of times the container gets called and the overall test success ratio. Given the obtained outcome, a satisfactory solution was not reached, and therefore, a new iteration of the action research methodology had to take place.

The first iteration and its constituent phases are summarized in Figure 19 in the form of a block diagram.

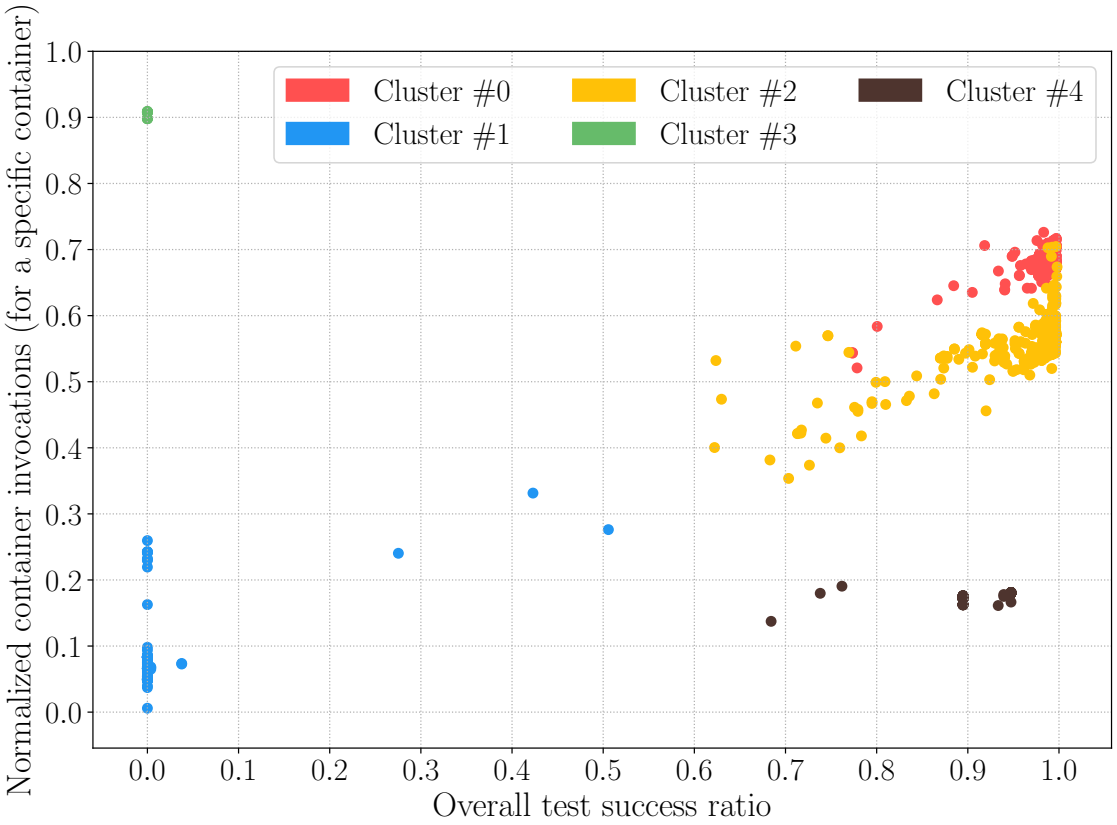
5.1.2 Second iteration: clustering and classification of the labeled log data

Seeing that no clear structure could be retrieved from the log data by clustering it, a new iteration of the action research methodology took place: the labeling of the data got taken into consideration.

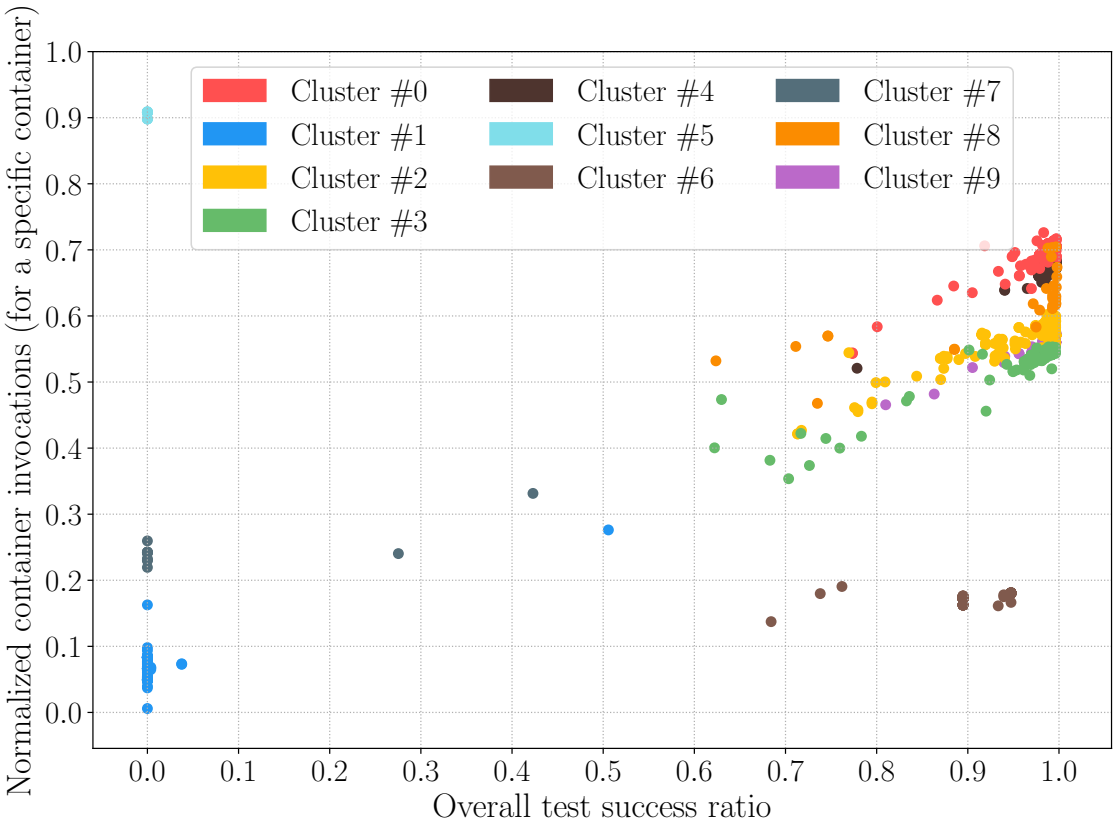
In the new phase of inquiry, one additional research question was included given that ground-truth root cause classes were pursued (which corresponds to the second question gathered in Section 1.3). The subsequent literature review focused on classification and clustering of labeled log data, and afterward, the design process established the short-term deadlines by the author and the thesis supervisors.

During the second phase of action, new logs were collected and added to the dataset by using the already-developed log collection software. The previously envisioned features were extracted (the feature extraction software remained unchanged). In order to get the data labeled, the author interviewed the testing engineers to retrieve the conceptual root cause categories that they conceptualize when dealing with failed tests. Five high-level descriptors were retrieved, which correspond to the most general root causes of failed tests (cf. Section 5.3). The testing engineers were then asked to manually label the collected test cases.

Having obtained labels, the classification MLP presented in Section 4.3 was now considered. While classification seems to be the most reasonable ML methodology for this new scenario, for the sake of completeness, clustering was also assessed so as to be able to quantify its performance and compare it to the results obtained with classification MLPs. Moreover, the data pre-processing techniques presented in Section 4.4.1 were now taken into consideration in order to assess the improvement obtained by applying more intricate variations of clustering algorithms to the original log data (cf. Section 5.4.1).

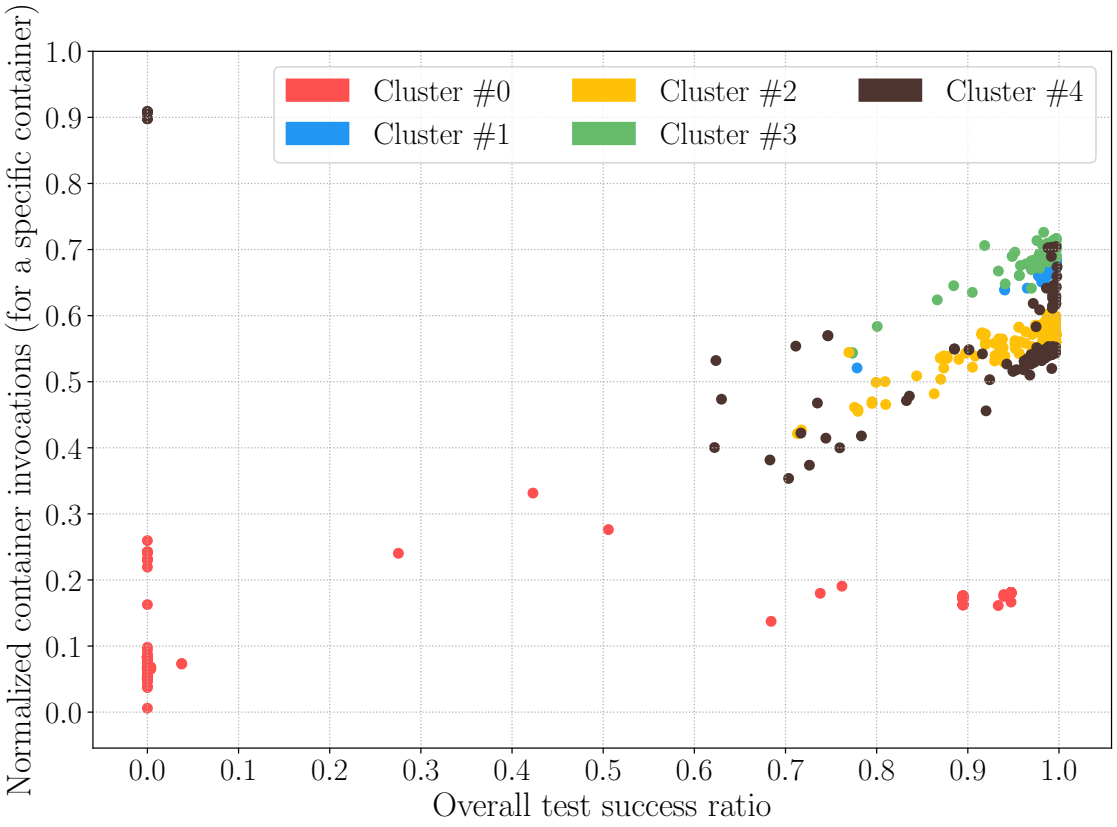


(a) k-means algorithm with $k = 5$.

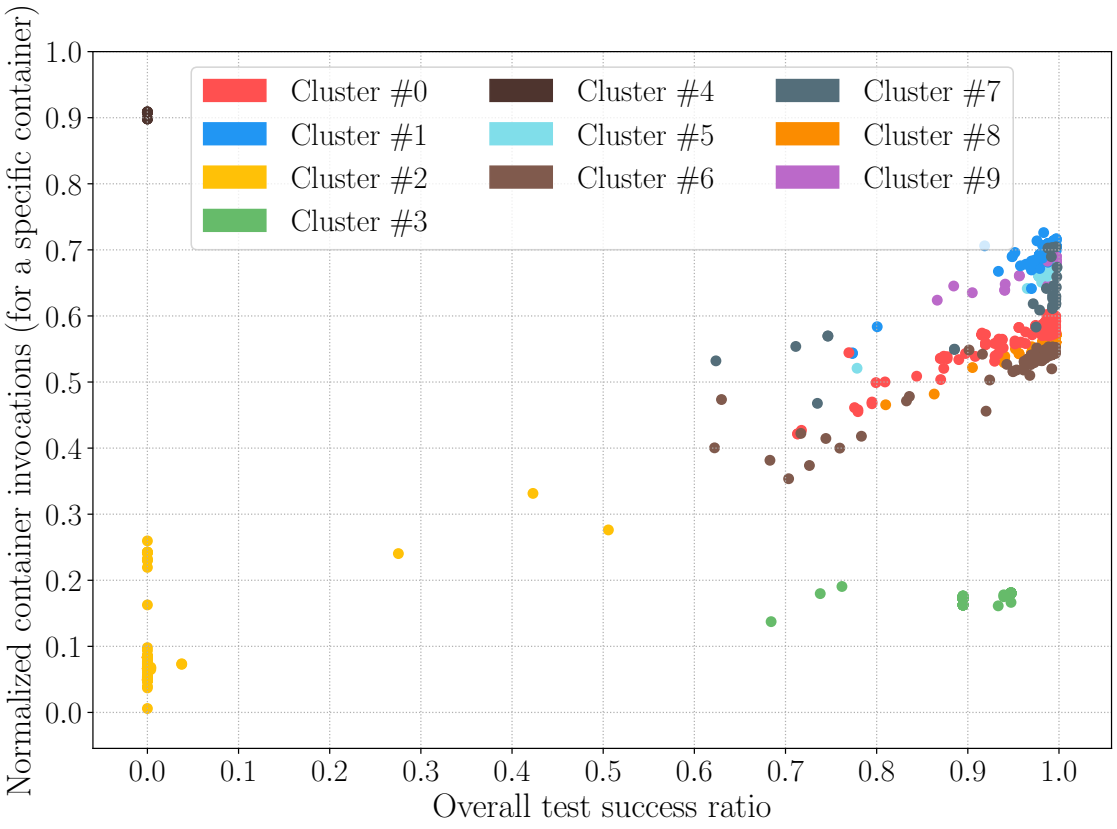


(b) k-means algorithm with $k = 10$.

Figure 17: Visualizations of the initial k-means-based clustering attempts.



(a) GMMs with $k = 5$.



(b) GMMs with $k = 10$.

Figure 18: Visualizations of the initial GMM-based clustering attempts.

In the third phase of analysis, the algorithms' outcomes were quantitatively assessed by means of two metrics: the adjusted mutual information (AMI) and the rand index (RI), which are further explained in Section 6.1.

Given that the obtained results were satisfactory (cf. Section 6), the fourth phase of conclusions took place, where the answers to the posed research questions were reached and reported (cf. Section 7).

In Figure 20, a block diagram displays the second iteration and its constituent phases.

5.2 Feature extraction

As has been mentioned at the beginning of this chapter, the feature extraction process is carried out by heuristics obtained from domain expert knowledge. The features are grouped into four distinct conceptual categories:

- **Container activity (count of entries)**: number of times a given container is invoked.
- **Server analytics (count of entries)**: HTTP 5xx responses, errors, tracebacks, and warnings.
- **Success rate per test suite**.
- **Overall test success**.

Table 2 lists the number of features in each group, as well as the normalization that is carried out for each one of them. The normalized features have their values bounded between 0 and 1 to ensure an equal contribution of each feature as well as to increase the speed in the convergence of the optimization algorithms [85, 69].

The success rates are obtained as the quotient of the number of passed tests divided by the number of scheduled tests, both for each suite and the global computation.

For the container invocations and server HTTP responses, intra-feature normalization is applied, meaning that each feature's count is divided by the maximum count within all examples.

As tests are being run, the number of features is subject to variation depending on which suites and containers are used in a given test pool. The feature extraction software is programmed to take this into account and to adapt the number of features dynamically as new tests are included (thus, the size of the feature groups in Table 2 that are followed by an asterisk (*) may change depending on the retrieved tests). With the final dataset, a total amount of 188 features is obtained.

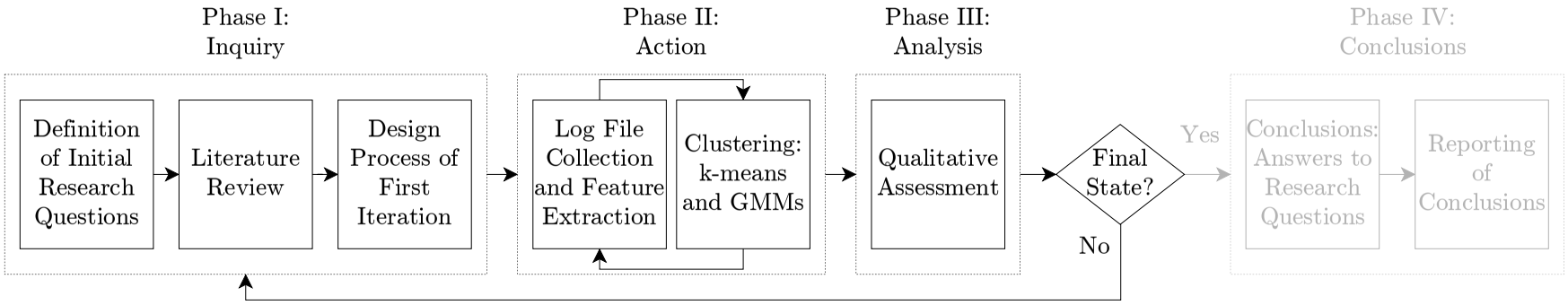


Figure 19: Block diagram describing the first iteration of the action research methodology carried out during the development of this Master's thesis (cf. Figure 1).

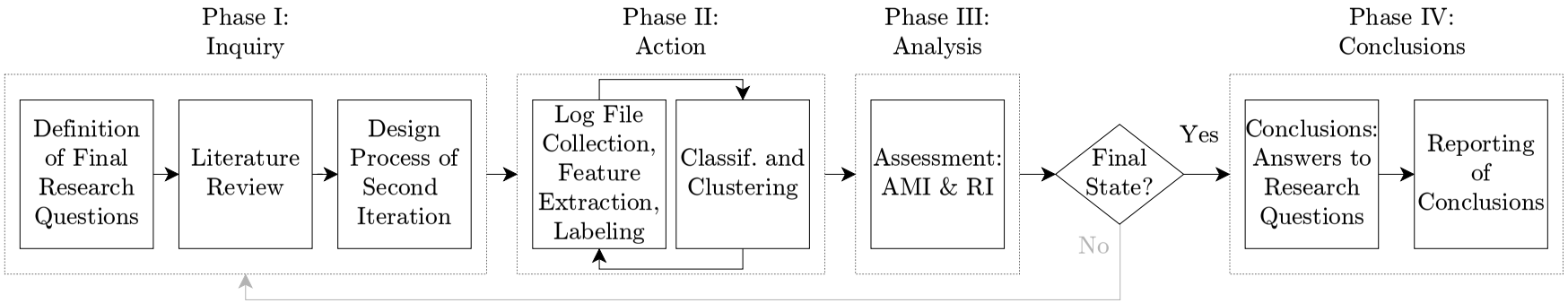


Figure 20: Block diagram describing the second iteration of the action research methodology carried out during the development of this Master's thesis (cf. Figure 1).

Feature category	Number of features	Feature scaling
Container invocations (normalized count of entries)	56*	Intra-feature normalization
Server analytics (normalized count of entries)	4	Intra-feature normalization
Success rate per test suite	127*	Ratio (passed/total)
Global success rate	1	Ratio (passed/total)

Table 2: Information on the extracted features.

5.3 Conceptual root cause classes

With the aim of automating the RCA carried out by the agile team at Ericsson Finland, the most general root cause categories that the testing engineers conceptualize are pursued. Five different classes are retrieved after interviewing the experts:

- **Functional errors**, where the error lies in the product's developed code (i.e., development errors).
- **Connectivity errors**, whose failure corresponds to a problem in the communication between the servers.
- **Infrastructure errors**, being the failure caused by the testing environment (e.g., hardware errors).
- **Test errors**, corresponding to failures caused by the presence of bugs in the tests' code.
- **Intentionally interrupted tests**, which gather tests whose execution was stopped by the testing engineers before completion.

5.4 Evaluated algorithms

The technologies presented in Section 4 have been deployed for the problem of accurately mapping the failed tests with their corresponding root causes: specifically, the k-means algorithm and GMMs for clustering and MLPs for classification (cf. Sections 4.3 and 4.4, respectively).

A total number of 1271 failed test cases have been collected and labeled, corresponding to the period ranging from February to July 2018.

These test cases are then split into train, validation and test sets at random, following a 60%/20%/20% division. 20 different random splits are formed with the

aforementioned ratios, so as to avoid overfitting to a given form of partitioning of the data. All algorithms are computed 20 times with these set divisions, and the average values and standard deviations of the evaluation metrics are reported (cf. Section 6).

5.4.1 Clustering analysis

The k-means algorithm is initialized 10 times with different random centroids, iterating at maximum 300 times, should it not converge before (the convergence being defined as no changes happening in the assignments). The best result from those 10 initializations is kept in terms of the pursued loss function (so that a suboptimal local minimum is avoided).

GMMs are also initialized 10 times, iterating a maximum number of 300 times with the same convergence criterion as k-means.

Given that the number of pursued root cause categories is 5, the number of clusters is set to be $k = 5$ (for the initial clustering attempts, more values of k, such as $k = 10$, are also considered).

Both algorithms are fed with several variations of the original feature matrix:

- **Raw data**, i.e., the feature matrix without any further alteration (with unlabeled and labeled data).
- **Data preprocessed with PCA**, keeping a number of principal components that retains 95% of the original variance on the training set (with labeled data).
- **Data preprocessed by an AE**, feeding the encoded representation of the original feature matrix. The most suitable ANN structure is found by means of hyperparameter tuning (grid search; with labeled data).

The AE's optimal hyperparameters are found via grid search: all combinations of architectures with a set of defined parameters are assessed, and the best-performing combination of hyperparameters is reported.

The considered structure of the AE is always undercomplete, where the number of hidden layers varies from 1 to 14. Specifically, the number of neurons is iterated to range all the powers of two from 1 to a number below the number of features. Given that the latter corresponds to 188, 128 is the last number of neurons being assessed.

The AE is designed to be symmetric; therefore, 7 different exponent values in both the encoder and the decoder are tested. All combinations of powers of two from 1 to 7 are constrained to be arranged in decreasing order in the encoder, and the structure is always mirrored in the decoder, leading to 28 total network structures.

Sigmoid, tanh, and ReLU are the evaluated activation functions, fulfilling that only one activation function is present in the whole network at a time. Both L1 and L2 regularization are carried out, each of them testing the regularization values of $\lambda = [0.01, 0.1, 1, 10]$, as well as no regularization.

SGD and Adam are the selected optimization algorithms, and a maximum number of 100000 epochs is run, where the early stopping criterion is used for determining the convergence of the validation loss (i.e., the training is stopped whenever the validation error starts to increase). Its patience is set to 50 epochs, which means that if the network's validation error does not decrease after 50 consecutive forward and backward passes, it stops training, and the ANN's weights at the last epoch that displayed improvement are kept. The learning rate α of SGD is set to 0.01, whereas in Adam, its value is equal to 0.001; the parameters β_1, β_2, and ε are set to 0.9, 0.999, and 1×10^{-8}, respectively, as recommended by the authors of the Adam algorithm [77].

Dropout regularization is also included, testing the ratios of 0.3, 0.5, and 0.8, as well as no dropout. Batch normalization is also tried out when no dropout is taking place, given the unsuitability of both methods acting simultaneously [86].

A standard AE is tested with the MSE loss function: the encoded representation it achieves is fed to the clustering algorithm. Additionally, the custom AE proposed by Song et al. [84] is implemented, whose loss function combines the standard MSE with a centroid updating term. For the latter case, the parameter controlling the contribution of the second term of the loss function is assessed with the values $\eta = [-1, -0.7, -0.5, -0.2, -0.1, 0.1, 0.2, 0.5, 0.7, 1]$.

The counts of all different AE hyperparameter possibilities are gathered in Tables 3 and 4 for the standard AE and the custom-loss AE proposal, respectively. The best results for each case are kept.

5.4.2 Classification

Regarding classification, MLPs are tested with hyperparameters similar to those that the AEs for clustering were assessed with: ReLU, sigmoid and tanh activation functions for the hidden layers (the activation function at the output layer always being softmax in order to scale the values to the $[0, 1]$ range); L1 and L2 regularization with $\lambda = [0.01, 0.1, 1, 10]$, and no regularization; Adam and SGD optimizers (with the same α, β_1, β_2, and ε parameters); 100000 epochs at maximum, making use of the early stopping criterion, with a patience equal to 50 epochs; dropout with ratios of 0.3, 0.5, 0.8, and no dropout, the last case with and without batch normalization. The used loss function is the categorical cross-entropy.

The first tested network structure contains a minimum of one hidden layer and a maximum of 10 hidden layers, whose number of neurons are placed in decreasing order. Powers of 2 are tested, whose exponents range from 1 to 10, as no undercomplete consideration is required for MLPs.

The count of hyperparameter permutations is shown in Table 5.

After this initial attempt, seeing that the best results consisted of a network with 7 hidden layers, a more extensive combination of network structures is designed,

collecting all permutations of values in decreasing order starting from 1000 neurons to 20 neurons with decrements of 70. The reason for the decrement value of 70 stems from the compromise between extensive testing and execution time.

The new count of hyperparameter permutations can be seen in Table 6.

5.4.3 Runtime

The vast number of runs were executed during 27 days on a dedicated machine, yielding the results gathered in Section 6.

Hyperparameters	**Number of runs**
Loss functions	1
Clustering models	2
Network structures	28
Optimizers	2
Activation functions	3
Dropout or batch norm.	5
Regularizer functions	2
Regularizer values	4
Total regularization combinations	$(4 \times 2) + 1 = 9$
Total count	**15120**

Table 3: Hyperparameter tuning for the standard AE: Itemized number of runs.

Hyperparameters	**Number of runs**
Loss functions	1
Weighing parameter η values	10
Clustering models	2
Network structures	28
Optimizers	2
Activation functions	3
Dropout or batch norm.	5
Regularizer functions	2
Regularizer values	4
Total regularization combinations	$(4 \times 2) + 1 = 9$
Total count	**151200**

Table 4: Hyperparameter tuning for the custom-loss AE: Itemized number of runs.

Hyperparameters	**Number of runs**
Loss functions	1
Network structures	55
Optimizers	2
Activation functions	3
Dropout or batch norm.	5
Regularizer functions	2
Regularizer values	4
Total regularization combinations	$(4 \times 2) + 1 = 9$
Total count	**14850**

Table 5: Hyperparameter tuning for the classification MLP: Itemized number of runs.

Hyperparameters	**Number of runs**
Loss functions	1
Network structures	6435
Optimizers	2
Activation functions	3
Dropout or batch norm.	5
Regularizer functions	2
Regularizer values	4
Total regularization combinations	$(4 \times 2) + 1 = 9$
Total count	**1737450**

Table 6: Hyperparameter tuning for the classification MLP (extensive analysis with 7 hidden layers): Itemized number of runs.

6 Results

6.1 Evaluation metrics

Given the need for quantifying the performance of the classification and clustering algorithms, the labels obtained from the developers' manual work are used for this purpose (cf. Section 5).

When assessing classification, the simple measure of accuracy or RI provides the ratio of correctly assigned data with respect to the total count of processed samples [87]:

$$\text{accuracy}\,(\Omega, \mathbb{C}) = \text{RI}\,(\Omega, \mathbb{C}) = \frac{1}{K}\sum_{i=1}^{k} |\omega_i \cap c_i|, \tag{38}$$

where $\mathbb{C} = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_k\}$ is the set of ground-truth classes, $\Omega = \{\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \ldots, \boldsymbol{\omega}_k\}$ is the set of predicted classes, k is the number of distinct ground-truth categories, and K is the total number of processed samples. Its value is bounded between 0 and 1, the former being a result of total mismatch, and the latter being the consequence of a perfect match.

Focusing the attention on clustering, the value the RI yields shows some dependency with the number of clusters and samples, a fact that can be corrected by introducing an adjustment for chance, leading to the adjusted RI (ARI) [88, 89, 90]:

$$\text{ARI}\,(\Omega, \mathbb{C}) = \frac{\text{RI}\,(\Omega, \mathbb{C}) - \text{E}\,[\text{RI}\,(\Omega, \mathbb{C})]}{\max\,(\text{RI}\,(\Omega, \mathbb{C})) - \text{E}\,[\text{RI}\,(\Omega, \mathbb{C})]}. \tag{39}$$

Purity is a straightforward measure that assesses the spread in the prediction of categories with respect to the ground-truth categories: it evaluates the tendency of clusters to contain a single ground-truth category [87]. For each of the obtained clusters, the most frequent ground-truth category is assigned to it, and then, its normalized sum is computed:

$$\text{purity}\,(\Omega, \mathbb{C}) = \frac{1}{K}\sum_{i=1}^{k} \max_{j} |\omega_i \cap c_j|. \tag{40}$$

Nonetheless, the measure of purity does not penalize the assignment with respect to the pursued ground-truth category for each cluster, but only considers the relative spread of assignments. It is bounded between $\frac{1}{k}$ and 1, the former representing the value of maximum spread in the assignments, and the latter indicating uniform assignments without any spread. Following a naive example, if a matching task assigns as many categories as available examples, the measure of purity will be equal

to 1, regardless of the number of ground-truth categories. This property confirms the need for more sophisticated metrics.

As an alternative, the mutual information (MI) measures the agreement or amount of information that the prediction and ground-truth assignments share, ignoring permutations [90]. Its computation is based on the concept of information or Shannon entropy (H), a measure of uncertainty of an RV V, defined as the expected value of its self-information (I), which is the surprise or uncertainty when sampling V [58, 23]:

$$\mathrm{H}(V) = \mathrm{E}[\mathrm{I}(V)] = \mathrm{E}\left[\log\left(\frac{1}{\mathrm{P}(V)}\right)\right] = \mathrm{E}\left[-\log(\mathrm{P}(V))\right] = -\sum_{i=1}^{n}\mathrm{P}(v_i)\log(\mathrm{P}(v_i)). \tag{41}$$

When the probability distribution of the RV V is close to uniform, the entropy is maximized (having a value close to 1), and the uncertainty when sampling the RV is high. When the probability distribution of V tends to be deterministic, the entropy is minimized (having a value close to 0), and the uncertainty when sampling V is low.

Conceptualizing the assignment for each element of the original dataset $\mathbb{D}$ to the ground-truth categories belonging to $\mathbb{C}$ as the RV A, and similarly, the matching from the dataset $\mathbb{D}$ to the predicted categories Ω as the RV B, their corresponding entropies are defined as:

$$\mathrm{H}(A) = -\sum_{i=1}^{N}\mathrm{P}(a_i)\log(\mathrm{P}(a_i)), \tag{42}$$

$$\mathrm{H}(B) = -\sum_{i=1}^{N}\mathrm{P}(b_i)\log(\mathrm{P}(b_i)), \tag{43}$$

where $\mathrm{P}(a_i) = \frac{|c_i|}{N}$ and $\mathrm{P}(b_i) = \frac{|\omega_i|}{N}$ are the probabilities of a randomly-picked object from A or B to belong to ground-truth category $\boldsymbol{c}_i$ or predicted category $\boldsymbol{\omega}_i$, respectively.

The MI between both RVs A and B is then defined as [79]:

$$\mathrm{MI}(A,B) = \sum_{i=1}^{N}\sum_{j=1}^{N}\mathrm{P}(a_i,b_j)\log\left(\frac{\mathrm{P}(a_i,b_j)}{\mathrm{P}(a_i)\,\mathrm{P}(b_j)}\right). \tag{44}$$

It represents the measure of mutual dependence between both RVs, i.e., how much information one RV conveys about the other one: the higher its value, the higher the dependence between the prediction and the ground-truth assignments.

The MI can be normalized with respect to the individual entropies, leading to the normalized MI (NMI), whose value is enclosed between 0 and 1:

$$\mathrm{NMI}\left(A,B\right)=\frac{\mathrm{MI}\left(A,B\right)}{\sqrt{\mathrm{H}\left(A\right)\mathrm{H}\left(B\right)}}. \tag{45}$$

As is the case with the RI, the values of MI increase in proportion to the number of clusters and samples, which leads to the adjusted MI or AMI [89, 90]:

$$\mathrm{AMI}\ \left(A,B\right)=\frac{\mathrm{MI}\ \left(A,B\right)-\ \mathrm{E}\ \left[\ \mathrm{MI}\ \left(A,B\right)\right]}{\max\left(\mathrm{H}\left(A\right),\mathrm{H}\left(B\right)\right)-\ \mathrm{E}\ \left[\ \mathrm{MI}\ \left(A,B\right)\right]}. \tag{46}$$

The AMI extends its validity to classification [91], and thus, it is also used for assessing classification algorithms.

6.2 Best-performing ANNs

When carrying out grid searches over the ANN structures presented in Section 5, the best results in terms of AMI are collected for each category of tested algorithms. Additionally, the values of accuracy are included in the classification algorithms given their intuitive meaning.

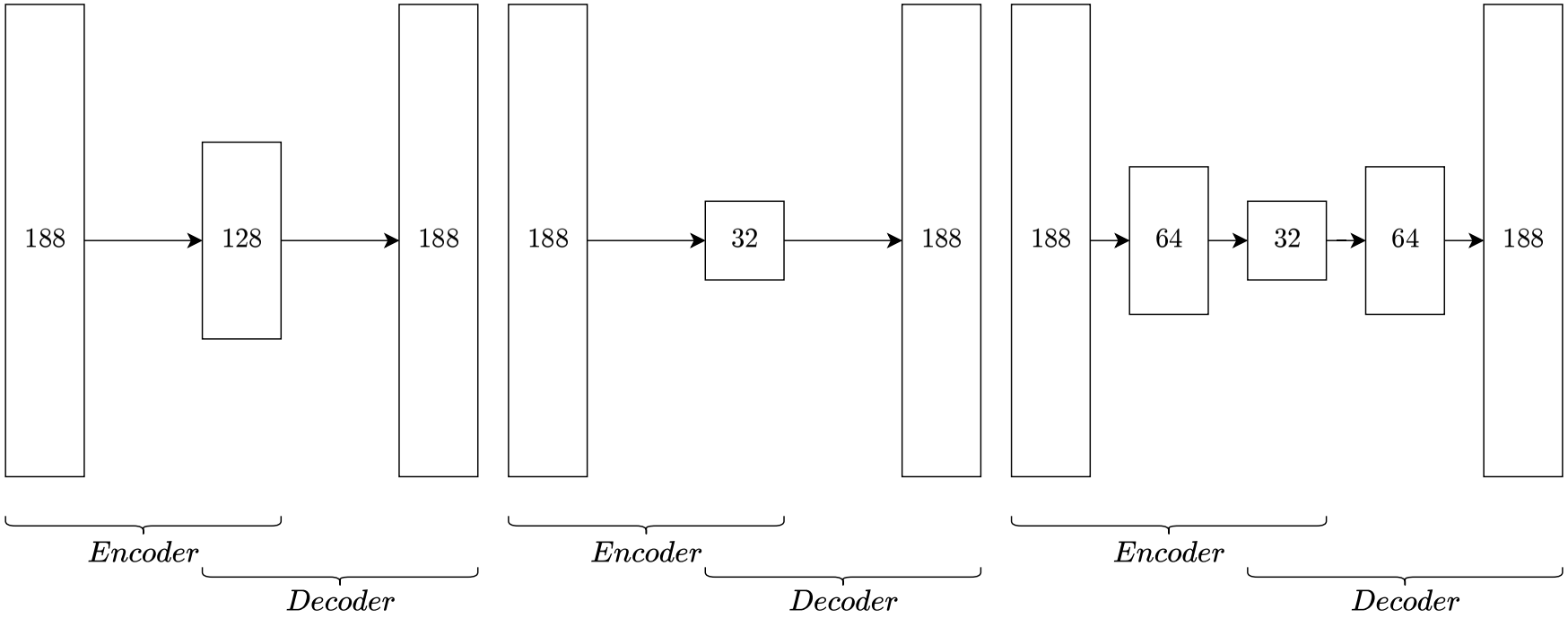
The ANNs' parameters are gathered in Table 7, and their visual structure can be found in Figures 21 and 22 for clustering and classification, respectively (where all the layers are depicted as rectangles with constant width and variable height, proportional to the number of neurons, shown as text inside the rectangles).

Concerning clustering, the best AEs display shallow architectures: but for the custom-loss case, where the number of hidden layers is equal to 3, the best AEs used in conjunction with k-means and GMMs have only one hidden layer, the former with 128 and the latter with 32 neurons. The dropout rate in both scenarios is 0.5; hence, no batch normalization is used. The AE that combines k-means with the custom loss consisting of the MSE and the centroid-update term has 3 hidden layers of sizes 64, 32 and 64, respectively. It neither regularizes with dropout nor implements batch normalization. The optimizers and activation functions for all AEs are Adam and ReLU, respectively.

Concerning classification, the best base-2 architecture has 7 hidden layers, and the extended architecture tests the same number of hidden layers but with different counts of neurons per layer (cf. Section 5.4). But for batch normalization, which the base-2 architecture does not implement (while the extended architecture does), all other parameters are shared in both networks: ReLU as an activation function, no dropout, and SGD as an optimizer.

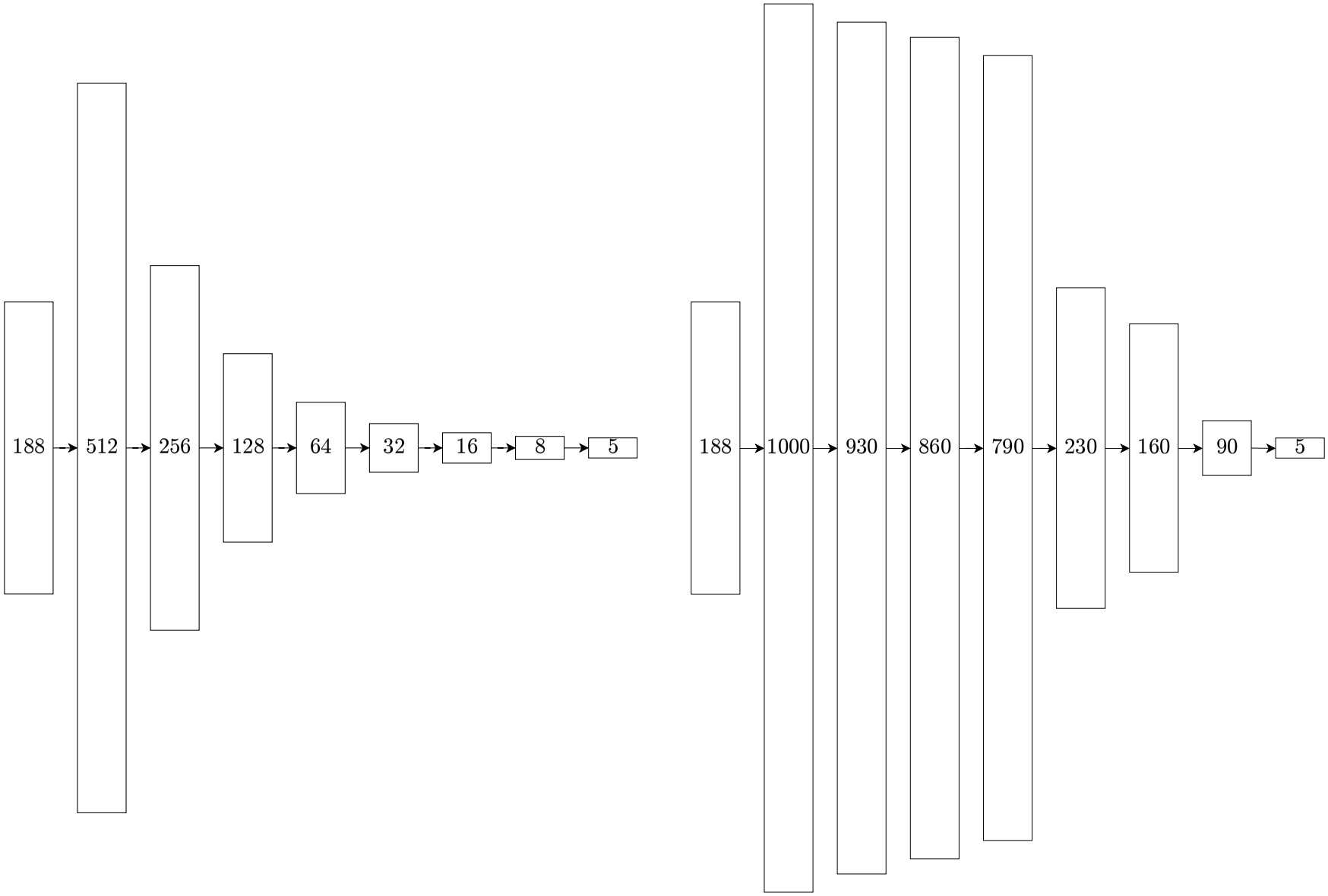
	Number of hidden layers	Structure of hidden layers	Activation functions in hidden layers	Dropout rate	Batch norm.	Optimizer
AE: k-means	1	[128]	ReLU	0.5	No	Adam
AE: GMMs	1	[32]	ReLU	0.5	No	Adam
AE: k-means (custom loss)	3	[64, 32, 64]	ReLU	0	No	Adam
MLP: base-2 architecture	7	[512, 256, 128, 64, 32, 16, 8]	ReLU	0	No	SGD
MLP: 7-layer architecture	7	[1000, 930, 860, 790, 230, 160, 90]	ReLU	0	Yes	SGD

Table 7: Best-performing ANN architectures.



(a) AE: k-means algorithm. (b) AE: GMMs. (c) AE: k-means algorithm (custom loss).

Figure 21: AE architectures for clustering.



(a) MLP: base-2 architecture.

(b) MLP: 7-layer extended architecture.

Figure 22: MLP architectures for classification.

6.3 Performance

The performance of the tested algorithms is shown in Tables 8 and 9.

Regarding clustering, applying PCA to the raw data does not cause any improvement with respect to clustering the raw data directly for both k-means and GMMs: the principal components are not able to find a better data representation. The results with GMMs are in fact worse than those obtained with k-means.

With the best AE, k-means still outperforms GMMs. There is a slight improvement when using k-means, and a greater one when using GMMs with respect to no pre-processing or to pre-processing with PCA, a fact that is expected due to the non-linear representation the AE obtains from the original data, which aids in the subsequent clustering.

Using the AE with the custom loss, the results reach the maximum average AMI value of 0.451, which proves that it is the most suitable solution for clustering: its loss function is defined in such a way that both actions of reconstruction and cluster assignment are pursued; hence, both actions achieve the best clusters.

However, classification feedforward networks significantly outperform clustering analysis, reaching an average accuracy value of 0.880 with the base-2 architecture (and an associated average AMI value of 0.654), and 0.889 with the extended architecture (where 0.665 is the associated average AMI value).

The final results prove that classification tasks are better suited than clustering-based approaches for this work's scenario.

	k-means	**GMMs**
Raw data	0.401 ± 0.050	0.295 ± 0.037
PCA (95% of original variance retained)	0.400 ± 0.052	0.290 ± 0.038
Clustering (via the standard AE)	0.415 ± 0.031	0.405 ± 0.036
Clustering (via the custom-loss AE)	0.451 ± 0.041	—

Table 8: Clustering AMI results.

	AMI	**Accuracy**
Base-2 architecture	0.654 ± 0.043	0.880 ± 0.021
7-layer architecture	0.665 ± 0.043	0.889 ± 0.020

Table 9: Classification AMI and accuracy results.

7 Conclusions

7.1 From expert knowledge to ML-based RCA

In this work, classification and clustering algorithms have been explored for accurately categorizing the root causes of failed tests in agile CI/CD software testing environments by means of log data analysis.

Clustering of the unlabeled data can be used as a tool for obtaining a first analysis of the log data: it can help retrieve an inherent structure in the data, should it exist. Unfortunately, the obtained results with this work's dataset did not correlate well with the root causes of the failed tests: while some clusters seemed to gather examples where a single failure condition was present, the meaning behind the other clusters remains vague. Therefore, for this scenario, clustering of the unlabeled data is not an efficient solution for automating the RCA of failed tests.

Labeling is an expensive process both in terms of time and manual labor; nonetheless, it is an efficient tool that enables the use of powerful supervised learning MLPs that achieve satisfactory results. For the sake of completeness, this thesis also evaluated clustering algorithms on the labeled log files, applied both to the raw data and to more complex pre-processed variations of it. Even though clustering on the more intricately pre-processed data improved the matching with the envisioned ground-truth categories with respect to applying simpler or no pre-processing techniques at all, classification MLPs outperform any other algorithm, and thus, they constitute the most efficient solution.

7.2 Answers to the research questions

Three research questions have been posed in this Master's thesis.

The extraction and definition of meaningful features from the testing log data is tackled by interviewing the testing engineers and by following their debugging activity, a process that requires a previous understanding of the testing environment. Even though not being a straightforward activity, by rigorously planning the interviews and following the debugging activity of the testing engineers in a structured fashion, a satisfactory collection of features can be obtained that best reflect the human expert knowledge used for RCA. As a result, four distinct conceptual feature groups are obtained that summarize the information the testing engineers make use of when dealing with manual RCA. Given the satisfactory results obtained with the classification MLPs, the envisioned feature groups adequately transfer the testing engineers knowledge to a feature matrix.

In order to map the existing developers' knowledge into distinctive ground-truth root-cause categories, further interviewing of the testing engineers has to take place, where the human experts are asked to frame the conceptual ground-truth categories they envision when dealing with failed tests. These interviews prove to be simpler than the ones that pursue crafting the feature matrix: the sought information is not latent, as the concept of "ground truth category" is used in their everyday debugging process.

Overall, these interviews establish a relationship between the information in the log data the human experts pay attention to and the root causes of the failed tests.

Lastly, concerning the use of ML for accurately identifying the root causes of the failed tests, this thesis experimentally shows how MLP-based classification proves to be the best-performing solution, significantly surpassing the action of clustering analysis (no matter how intricate its foundation): the best ANN correctly assigns an example with its failure root cause, on average, 88.9% of the time.

7.3 Promising future research directions

The results of the best-performing MLP solution can be further improved by continuing the development and research on several areas:

- A more extensive hyperparameter optimization could be run by making use of a parallelized implementation running on several machines, searching over a broader range of hyperparameters.
- With respect to the ground-truth failure categories, more granularity could be considered, taking as labels more specific categories than the highest-level ones. For this to be achieved, the testing engineers would need to be interviewed more extensively, and a diagram with their envisioned ground-truth categories would need to be constructed.
- The extraction of new features might improve the algorithms' outcome, whose envisioning could be carried out by means of extensive interviewing of the testing engineers; in fact, the feature extraction software is already programmed to extract an additional feature group with values related to CPU usage in the servers that run the tests, but its inclusion had to be discarded in this work due to the earliest tests not having this information available.
- The use of natural language processing could be tested for mining the log files directly and obtaining a representation in a new feature space, which could then be used for classification, as presented by Bertero et al. [92]. Additionally, more research could be carried out in this area with the aim of finding more intricate proposals, and their performance could be compared to traditional classification with a set of defined features, as has been carried out in this work.

All in all, the performance of the system might be enhanced by both expanding the number of ground-truth categories and the collection of features and by adding complexity to the tested algorithms.

The work carried out in this thesis gathers a first approach towards automating test failure RCA in an agile CI/CD software testing environment. A manual revision of the best-performing MLP's outcome is required; nonetheless, the aforementioned improvement areas might make the machine's output less dependent on an *a posteriori* human validation.

References

[1] R. R. Schaller, "Moore's law: past, present and future", *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, June 1997.

[2] M. Castells, "An Introduction to the Information Age", *City*, vol. 2, no. 7, pp. 6–16, May 1997.

[3] M. Campbell-Kelly, *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*, 1st ed. Cambridge, MA, USA: MIT Press, February 2004.

[4] I. Sommerville, *Software Engineering*, 9th ed., ser. International Computer Science. Wokingham, UK: Addison-Wesley, March 2010.

[5] Software.org: the BSA Foundation, "The Growing $1 Trillion Economic Impact of Software", Software.org: the BSA Foundation, Washington, DC, USA, Tech. Rep., September 2017, available at: https://software.org/wp-content/uploads/2017_Software_Economic_Impact_Report.pdf. Accessed: 2018-12-20.

[6] D. Gelperin and B. Hetzel, "The Growth of Software Testing", *Communications of the ACM*, vol. 31, no. 6, pp. 687–695, June 1988.

[7] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Hoboken, NJ, USA: John Wiley & Sons, November 2011.

[8] J. H. Andrews, "Testing using Log File Analysis: Tools, Methods, and Issues", in *Proceedings of the 13th Annual International Conference on Automated Software Engineering (ASE '98)*. Honolulu, HI, USA: IEEE, October 1998, pp. 157–166.

[9] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*, 1st ed. Reading, MA, USA: Addison-Wesley Professional, July 1999.

[10] J. J. Rooney and L. N. V. Heuvel, "Root Cause Analysis for Beginners", *Quality Progress*, vol. 37, no. 7, pp. 45–53, July 2004.

[11] R. J. Latino, K. C. Latino, and M. A. Latino, *Root Cause Analysis: Improving Performance for Bottom Line Results*, 4th ed. Boca Raton, FL, USA: CRC Press, July 2011.

[12] T. O. Lehtinen, M. V. Mäntylä, and J. Vanhanen, "Development and evaluation of a lightweight root cause analysis method (ARCA method) – Field studies at four software companies", *Information and Software Technology*, vol. 53, no. 10, pp. 1045–1061, October 2011.

[13] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams", in *Proceedings of the 1st Workshop on Future of Software Engineering (FOSE '07) at ICSE 2007*. Los Alamitos, CA, USA: IEEE Computer Society, May 2007, pp. 85–103.

[14] G. G. Claps, R. B. Svensson, and A. Aurum, "On the Journey to Continuous Deployment: Technical and Social Challenges along the Way", *Information and Software Technology*, vol. 57, pp. 21–31, January 2015.

[15] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*, 1st ed., ser. Addison-Wesley Signature Series. Boston, MA, USA: Pearson Education, July 2007.

[16] W. Li, "Automatic Log Analysis using Machine Learning: Awesome Automatic Log Analysis version 2.0", Master's thesis, Uppsala University, Uppsala, Sweden, November 2013.

[17] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing", National Institute of Standards and Technology, RTI Project Number 7007.011, Research Triangle Park, NC, Tech. Rep., May 2002, available at: https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf. Accessed: 2018-12-20.

[18] A. Oliner and J. Stearley, "What Supercomputers Say: A Study of Five System Logs", in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*. Edinburgh, UK: IEEE/IFIP, June 2007, pp. 575–584.

[19] T. M. Mitchell, *Machine Learning*, 1st ed., ser. Computer Science. New York, NY, USA: McGraw-Hill Education, March 1997.

[20] H. Lune and B. L. Berg, *Qualitative Research Methods for the Social Sciences*, 9th ed. Boston, MA, USA: Pearson Education, January 2017.

[21] E. T. Stringer, *Action Research*, 3rd ed. Thousand Oaks, CA, USA: Sage Publications, May 2007.

[22] M. Fowler and J. Highsmith, "The Agile Manifesto", *Software Development Magazine*, vol. 9, no. 8, pp. 28–35, August 2001.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, 1st ed. Cambridge, MA, USA: MIT Press, November 2016, available at: http://www.deeplearningbook.org. Accessed: 2018-12-20.

[24] SAS Institute, "Machine Learning: What it is and why it matters", Available at: https://www.sas.com/en_us/insights/analytics/machine-learning.html, accessed: 2018-12-20.

[25] dimensions.ai, "dimensions.ai: Worldwide publication count for "machine learning", "deep learning", and "data science"", available at: https://app.dimensions.ai/analytics/publication/viz/overview-publications?search_text=%22machine%20learning%22&search_type=kws&search_field=full_search, https://app.dimensions.ai/analytics/publication/viz/overview-publications?search_text=%22deep%20learning%22&search_type=kws&search_field=full_search, https://app.dimensions.ai/analytics/

publication/viz/overview-publications?search_text=%22data%20science%22&search_type=kws&search_field=full_search, accessed: 2018-12-20.

[26] J. Clark, "Why 2015 Was a Breakthrough Year in Artificial Intelligence", available at: https://www.bloomberg.com/news/articles/2015-12-08/why-2015-was-a-breakthrough-year-in-artificial-intelligence, accessed: 2018-12-20.

[27] A. Denise, M.-C. Gaudel, and S.-D. Gouraud, "A Generic Method for Statistical Testing", in *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE '04)*. Saint-Malo, France: IEEE Computer Society, November 2004, pp. 25–34.

[28] A. X. Zheng, "Statistical Software Debugging", Ph.D. dissertation, University of California, Berkeley, Berkeley, CA, USA, December 2005.

[29] N. Baskiotis, M. Sebag, M.-C. Gaudel, and S.-D. Gouraud, "A Machine Learning Approach for Statistical Software Testing", in *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI '07)*. Hyderabad, India: International Joint Conferences on Artificial Intelligence, January 2007, pp. 2274–2279.

[30] L. C. Briand, "Novel Applications of Machine Learning in Software Testing", in *Proceedings of the 8th International Conference on Quality Software (QSIC '08)*. Oxford, UK: IEEE, August 2008, pp. 3–10.

[31] M. Noorian, E. Bagheri, and W. Du, "Machine Learning-based Software Testing: Towards a Classification Framework", in *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE '11)*. Miami Beach, FL, USA: Knowledge Systems Institute, July 2011, pp. 225–229.

[32] J. Kim, J. W. Ryu, H.-J. Shin, and J.-H. Song, "Machine Learning Frameworks for Automated Software Testing Tools: A Study", *International Journal of Contents*, vol. 13, no. 1, pp. 38–44, March 2017.

[33] dimensions.ai, "dimensions.ai: Worldwide publication count for "machine learning" AND "software testing" and "deep learning" AND "software testing"", available at: https://app.dimensions.ai/analytics/publication/viz/overview-publications?search_text=%E2%80%9Cmachine%20learning%E2%80%9D%20AND%20%E2%80%9Csoftware%20testing%E2%80%9D&search_type=abs, https://app.dimensions.ai/analytics/publication/viz/overview-publications?search_text=%E2%80%9Cdeep%20learning%E2%80%9D%20AND%20%E2%80%9Csoftware%20testing%E2%80%9D&search_type=abs, accessed: 2018-12-20.

[34] G. M. Weiss and H. Hirsh, "Learning to predict rare events in event sequences", in *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD '98)*. New York, NY, USA: American Association for Artificial Intelligence, August 1998, pp. 359–363.

[35] R. Vaarandi, “A Data Clustering Algorithm for Mining Patterns From Event Logs”, in *Proceedings of the 3rd IEEE Workshop on IP Operations and Management (IPOM 2003)*. Kansas City, MO, USA: IEEE, October 2003, pp. 119–126.

[36] E. W. Fulp, G. A. Fink, and J. N. Haack, “Predicting Computer System Failures Using Support Vector Machines”, in *Proceedings of the USENIX Workshop on the Analysis of System Logs (WASL ’08)*. San Diego, CA, USA: USENIX, December 2008, pp. 5–5.

[37] F. Salfner, “Event-based Failure Prediction: An Extended Hidden Markov Model Approach”, Ph.D. dissertation, Humboldt University of Berlin, Berlin, Germany, February 2008.

[38] I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko, “Failure prediction based on log files using Random Indexing and Support Vector Machines”, *Journal of Systems and Software*, vol. 86, no. 1, pp. 2–11, January 2013.

[39] W. Lee, “Applying Data Mining to Intrusion Detection: the Quest for Automation, Efficiency, and Credibility”, *SIGKDD Explorations*, vol. 4, no. 2, pp. 35–42, December 2002.

[40] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting Large-Scale System Problems by Mining Console Logs”, in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. Big Sky, MT, USA: ACM, October 2009, pp. 117–132.

[41] Alex Zhitnitsky, “Splunk vs ELK: The Log Management Tools Decision Making Guide”, available at: https://blog.takipi.com/splunk-vs-elk-the-log-management-tools-decision-making-guide/, accessed: 2018-12-20.

[42] Splunk Inc., “Splunk Inc. Announces Fiscal First Quarter 2019 Financial Results”, available at: https://www.splunk.com/en_us/newsroom/press-releases/2018/splunk-inc-announces-fiscal-first-quarter-2019-financial-results.html, accessed: 2018-12-20.

[43] Elasticsearch BV, “Elasticsearch Grows to 6 Million Downloads, Adds Former Box SVP as CMO”, available at: https://www.elastic.co/fr/blog/press/elasticsearch-grows-6-million-downloads-adds-former-box-svp-cmo, accessed: 2018-12-20.

[44] J. Stearley, S. Corwell, and K. Lord, “Bridging the Gaps: Joining Information Sources with Splunk”, in *Proceedings of the USENIX Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML ’10)*. Vancouver, BC, Canada: USENIX, October 2010, pp. 8–8.

[45] Google LLC, “Google Trends: Worldwide normalized search volume for "elasticsearch+logstash+kibana" and "splunk"”, available at:

https://trends.google.com/trends/explore?date=all&q=elasticsearch%20%2B%20logstash%20%2B%20kibana,splunk&hl=en-US, accessed: 2018-12-20.

[46] J. Stearley, "Towards Informatic Analysis of Syslogs", in *Proceedings of the 5th IEEE International Conference on Cluster Computing (CLUSTER 2004)*. San Diego, CA, USA: IEEE, September 2004, pp. 309–318.

[47] I. Rigoutsos and A. Floratos, "Combinatorial pattern discovery in biological sequences: The TEIRESIAS algorithm", *Bioinformatics*, vol. 14, no. 1, pp. 55–67, February 1998.

[48] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning", in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Dallas, TX, USA: ACM, November 2017, pp. 1285–1298.

[49] B. Debnath, M. Solaimani, M. A. G. Gulzar, N. Arora, C. Lumezanu, J. Xu, B. Zong, H. Zhang, G. Jiang, and L. Khan, "LogLens: A Real-time Log Analysis System", in *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS 2018)*. Vienna, Austria: IEEE, July 2018, pp. 1052–1062.

[50] M. Felldin, "Machine Learning Methods for Fault Classification", Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, December 2014.

[51] SolutionsIQ, Inc., "Integration Hell", Available at: https://www.solutionsiq.com/agile-glossary/integration-hell/, accessed: 2018-12-20.

[52] C. Pahl, "Containerisation and the PaaS Cloud", *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, July 2015.

[53] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes", *IEEE Cloud Computing*, no. 3, pp. 81–84, September 2014.

[54] Docker, Inc., "What is a Container", available at: https://www.docker.com/what-container, accessed: 2018-12-20.

[55] P. Bourque and R. E. Fairley, *Guide to the software engineering body of knowledge (SWEBOK (R))*, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society, January 2014.

[56] Microsoft Corporation Developer Network, "Test Cases and Test Suites", available at: https://msdn.microsoft.com/en-us/library/ee620496.aspx, accessed: 2018-12-20.

[57] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", Internet Requests for Comments, Internet Engineering Task Force, Fremont, CA, USA, RFC 7231, June 2014, available at: https://www.ietf.org/rfc/rfc7231.txt. Accessed: 2018-12-20.

[58] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, December 2009.

[59] D. L. Poole, A. K. Mackworth, and R. Goebel, *Computational Intelligence: A Logical Approach*, 1st ed. New York, NY, USA: Oxford University Press, January 1998.

[60] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society*, vol. 2, no. 1, pp. 230–265, January 1937.

[61] D. R. Hofstadter, *Gödel, Escher, Bach*, 1st ed. New York, NY, USA: Basic Books, April 1979.

[62] R. C. Schank, "Where's the AI?", *AI magazine*, vol. 12, no. 4, pp. 38–49, December 1991.

[63] R. Kurzweil, *The Age of Spiritual Machines: When Computers Exceed Human Intelligence*, 1st ed. London, UK: Penguin Books, January 2000.

[64] ——, *The Singularity Is Near: When Humans Transcend Biology*. London, UK: Penguin Books, September 2006.

[65] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon, "A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, August 31, 1955", *AI magazine*, vol. 27, no. 4, pp. 12–14, December 2006.

[66] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers", *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, July 1959.

[67] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane, "Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming", in *Artificial Intelligence in Design '96*, 1st ed. Dordrecht, Netherlands: Kluwer Academic Publishers, May 1996, pp. 151–170.

[68] O. Chapelle, B. Scholkopf, and A. Zien, *Semi-Supervised Learning*, 1st ed., ser. Adaptive Computation and Machine Learning Series. Cambridge, MA, USA: MIT Press, September 2006.

[69] A. Jung, "Machine Learning: Basic Principles", arXiv preprint, available at: https://arxiv.org/abs/1805.05052v8, October 2018, accessed: 2018-12-20.

[70] S. O. Haykin, *Neural Networks and Learning Machines*, 3rd ed. Upper Saddle River, NJ, USA: Pearson, November 2009.

[71] J. O. Berger, *Statistical Decision Theory and Bayesian Analysis*, 2nd ed., ser. Springer Series in Statistics. New York, NY, USA: Springer Science+Business Media, August 1985.

[72] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed., ser. Springer Series in Statistics. New York, NY, USA: Springer Science+Business Media, February 2009.

[73] S. Ruder, "An overview of gradient descent optimization algorithms", arXiv preprint, available at: https://arxiv.org/abs/1609.04747v2, June 2017, accessed: 2018-12-20.

[74] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks", in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*, vol. 15. Fort Lauderdale, FL, USA: Society for Artificial Intelligence and Statistics, April 2011, pp. 315–323.

[75] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, pp. 436–444, May 2015.

[76] P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences", Ph.D. dissertation, Harvard University, Cambridge, MA, USA, August 1974.

[77] D. P. Kingma and J. L. Ba, "Adam: A Method for Stochastic Optimization", in *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*. San Diego, CA, USA: International Conference on Representation Learning, May 2015.

[78] G. Cybenko, "Approximation by Superpositions of a Sigmoidal Function", *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, December 1989.

[79] K. Murphy and F. Bach, *Machine Learning: A Probabilistic Perspective*, 1st ed., ser. Adaptive Computation and Machine Learning. Cambridge, MA, USA: MIT Press, August 2012.

[80] H. W. Kuhn, "The Hungarian method for the assignment problem", *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, March 1955.

[81] I. Jolliffe, *Principal Component Analysis*, 2nd ed., ser. Springer Series in Statistics. New York, NY, USA: Springer Science+Business Media, October 2002.

[82] N. O'Rourke and L. Hatcher, *A Step-by-Step Approach to Using SAS for Factor Analysis and Structural Equation Modeling*, 2nd ed. Cary, NC, USA: SAS Institute, March 2013.

[83] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks", *Science*, vol. 313, no. 5786, pp. 504–507, July 2006.

[84] C. Song, F. Liu, Y. Huang, L. Wang, and T. Tan, "Auto-encoder Based Data Clustering", in *Proceedings of the 18th Iberoamerican Congress on Pattern Recognition (CIARP 2013)*. Havana, Cuba: Springer, November 2013, pp. 117–124.

[85] S. Raschka, "About Feature Scaling and Normalization (and the effect of standardization for Machine Learning algorithms)", available at: https://sebastianraschka.com/Articles/2014_about_feature_scaling.html, accessed: 2018-12-20.

[86] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift", arXiv preprint, available at: https://arxiv.org/abs/1801.05134v1, January 2018, accessed: 2018-12-20.

[87] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, 1st ed. Cambridge, UK: Cambridge University Press, July 2008.

[88] L. Hubert and P. Arabie, "Comparing partitions", *Journal of Classification*, vol. 2, no. 1, pp. 193–218, December 1985.

[89] N. X. Vinh, J. Epps, and J. Bailey, "Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance", *Journal of Machine Learning Research*, vol. 11, pp. 2837–2854, October 2010.

[90] Scikit-learn, "Clustering performance evaluation", available at: http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation, accessed: 2018-12-20.

[91] H. Bao-Gang, H. Ran, and Y. Xiao-Tong, "Information-theoretic Measures for Objective Evaluation of Classifications", *Acta Automatica Sinica*, vol. 38, no. 7, pp. 1169–1182, July 2012.

[92] C. Bertero, M. Roy, C. Sauvanaud, and G. Trédan, "Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection", in *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE 2017)*. Toulouse, France: IEEE, October 2017, pp. 351–360.